Chapter 1

Introduction

An algorithm, named for the ninth century persian mathematician al-khowarizmi is simply a set of rules used to perform some calculations, either by hand or more usually on a machine. In this subject. We will refer algorithm as a method that can be used by the computer for solution of a problem. Use of algorithm for some computations is a common practice. For instance performing the addition, multiplication is an algorithm.

Basically algorithm is a finite set of instructions that can be used to perform certain task. In this chapter. We will learn some basic concepts of algorithm.

INSIDE THIS CHAPTER

1.1. ALGORITHM

In this section, we will first understand, "what is algorithm?" and "when it is required?"

Definition of Algorithm

The Algorithm is defined as a collection of unambiguous instructions occuring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

This definition of algorithm is represent in Fig. 1.1.

After understanding the problem statement, we have to create an algorithm carefully for the given problem. The algorithm is then converted into some programming language and then given to same computing device *i.e.* computer. The computer then executes this algorithm which is actually submitted in the form of source program. During the process of execution it requires certain set of inputs with the help of algorithm and input set, the result is produced as an output. If the given input is invalid then it should raise appropriate error message; otherwise correct result will be produced as an output.

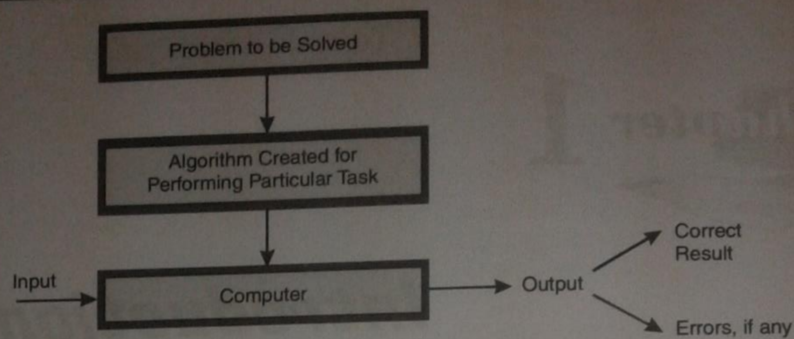


Fig. 1.1. Notion of Algorithm

1.2. PROPERTIES OF ALGORITHM

Simply writing the sequence of instructions as an algorithm is not sufficient to accomplis certain task. It is necessary to have following properties associated with an algorithm :

(*i*) **Non-ambiguity :** Each step in an algorithm should be non-ambiguous. That means eac instruction should be clear and precise. The instruction in an algorithm should not denot any conflicting meaning. This property also indicate the effectiveness of algorithm.

(*ii*) **Range of Input :** The range of input should be specified. This is because normally th algorithm is input driven and if the range of the input is not been specified then algorith can go in an infinite state.

(*iii*) **Multiplicity :** The same algorithm can be represented in several different ways. Th means, we can write in simple english the sequence of instructions or we can write it in t form of pseudo code. Similarly for solving the same problem we can write several differe algorithms. For instance : For searching a number from the given list, we can use sequenti or binary search. Here searching is a task and can use either a "sequential search" or "bina serach".

(*iv*) **Speed :** The algorithm are written using some specific ideas (which is popularly kno as logic of algorithm). But such algorithms should be efficient and should produce t output with fast speed.

(*v*) **Finiteness :** The algorithm should be finite. That means after performing requir operations it should terminate.

1.3. ISSUES IN WRITING AN ALGORITHM

There are various issues in the study of algorithm and those are :

(i) How to Devise Algorithm ?

The creation of an algorithm is alogical activity and one can not automate it . But there certain algorithmic design strategies and using these strategies one can create many useful algorit Hence mastering of such design strategies is an important activity in study of design and analysi algorithm.

(ii) How to Validate Algorithm ?

The next step after creation of algorithm is to validate it. The process of checking whether an algorithm computes the correct answer for all possible legal inputs is called algorithm validation. The purpose of validation is to find whether algorithm works properly without being dependent upon programming languages.

(iii) How to Analyze Algorithm ?

Analysis of algorithm is a task of determining how much computing time and storage is required by an algorithm. Analysis of algorithm is also called performance analysis and it is based on mathematics and a judgement is often needed about better algorithm when two algorithm get compared.

The behaviour of algorithm in best case, worst can and average case needs to be obtained.

(iv) How to Test a Program ?

After finding an efficient algorithm, it is necessary to test that the program written using the efficient algorithm behave properly or not. Testing of a program is an activity that can be carried out in two phases :

(*i*) Debugging and (*ii*) Performance measuring. While debugging a program, it is checked whether program produces faulty results for valid set of input, and if it is found then the program has to be corrected. Thus by debugging throughly the program is corrected profiling or performance is a process of measuring time and space required by a corrected program for valid set of inputs.

1.4. HOW TO WRITE AN ALGORITHM ?

Algorithm is basically a sequence of instructions written in simple English language. The algorithm is broadly divided into two section :

Algorithm Heading
It consists of name of algorithm, problem description. i/p and o/p.

Algorithm Body
It consists of logical body by making use of various programming constructs and assignment starts.

In this section, we will discuss the notations used for algorithm that are covered below :

(*i*) An algorithm is described within a pair of lines :

```
Procedure Name
__________
__________
End Name
```

(*ii*) Assignment is denoted by

```
Variable name ← expression
```

(*iii*) The symbol ∇ indicates comments.

(*iv*) Arrow in both direction ↔ is used to show exchange.

(*v*) If condition is shown as :

```
If condn then
   stmt (s);
else
   stmt (s);
end if;
```

(*vi*) Do-loop is shown as :

```
For variable = Initial value to final value {step step-label}
DO
____________
____________
END DO
```

(*vii*) Comment is shown as :

```
/* start */
```

(*viii*) While construct is used as :

```
While condn DO
____________
____________
End while
```

(*ix*) Case construct is used as :

```
Case
____________
____________
End Case
```

(*x*) Length [A] shows length of array.

(*xi*) $A[i.....j]$ shows array from i to j.

(*xii*) Arithematic operators +, –, *, / and % are used.

(*xiii*) Returning result is shown as :

```
Return (value (s))
```

Example 1.1. To understand the expressing method. Lets consider an algorithm which sum '*n*' numbers.

Algorithm 1.1. Without pseudo code

Step 1 : Select n numbers.

Step 2 : Set sum S to zero.

Step 3 : Repeat from first number to nth number *i.e.*

$$S = S + A[i]$$

Step 4 : Return sum (S).

Now we give the same algorithm with pseudo code :

```
Sum (A, n)
1. S ← 0
2. For i ← 1 to n
3. S ← S + A[i]
4. return S.
```

Now, we will again give algorithm for sum but this algorithm is iterative algorithm (recursive sum).

```
Rec_sum (A, n)
 1. S ← 0
 2. if (n ≤ 0) then
 3. return 0
 4. else
 5. return Rec_sum (A, n - 1) + A[n]
```

1.5. CORRECTNESS OF AN ALGORITHM

For any algorithm we must prove that it always return the desired output for all legal instances of the problem.

Example : For sorting, this means if :

(*a*) The input is already sorted.

or (*b*) It contains repeated elements.

Correctness is not Obvious

Suppose you have a robot arm equipped with a tool. Say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points. So, the robot visits the first contact point, then visit the second point, third and so forth until the job is done.

Since, the robot are expensive, we need to find the order, which minimizes the time (*i.e.*, travel distance) it takes to assemble the circuit board.

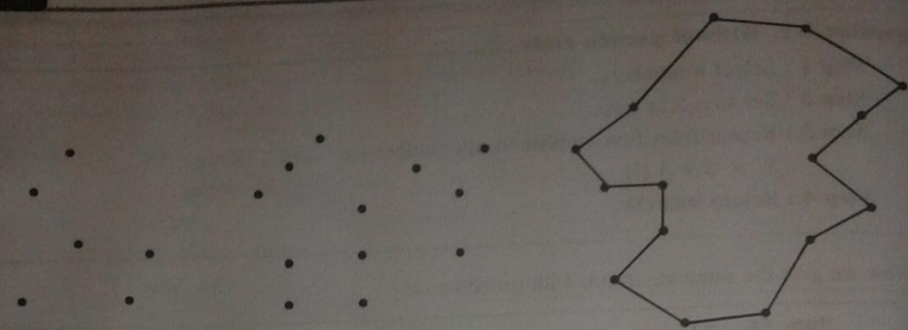


Fig. 1.2. Assembling of circuit board.

Now, the problem arises is that to find an algorithm for best tour to assemble the whole points for the circuit board.

1.6. ANALYSIS OF ALGORITHM

Analysis of algorithm means developing a formula or prediction of how fast the algorithm works bassed on the problem size.

The problem size could be :

(*i*) The number of inputs/outputs in an algorithm.

(*ii*) The number of operations involved in an algorithms.

The analysis of algorithm based on time of computation is called as time complexity of the algorithm.

Analysis of algorithm is also defined as the process of determing a formula for prediction of the memory requirement (primary memory) based on the problem size called space complexity of the algorithm.

Analyzing of an algorithm is concerned of three cased :

1. Worst case complexity
2. Best case complexity
3. Average case complexity

1. Worst Case Complexity : The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

2. Best Case Complexity : The best case complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.

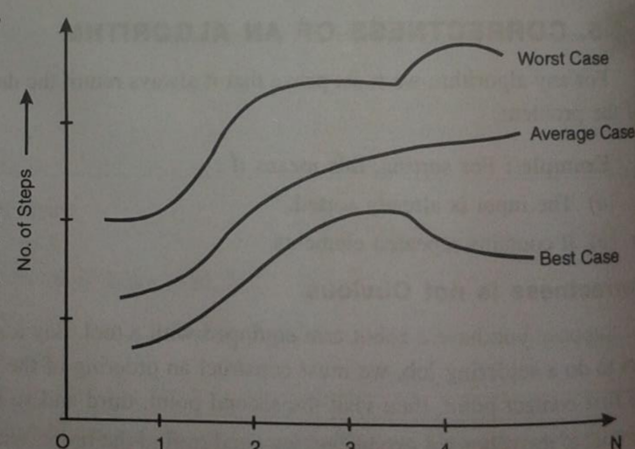


Fig. 1.3.

3. Average Case Complexity : The average case complexity of an algorithm is the function defined by the average number of steps taken on an instance of size n.

Each of these function defines a numerical function

(*a*) Time

(*b*) Space

Example 1.2. To understand the analysis of the algorithm let us consider an example of "insertion sort".

One way to sort an array of 'n' elements is to start with an empty list, then successively insert new elements in the proper positions.

$$a_1 \le a_2 \le a_3 \le .... \le a_n$$

At each stage, the inserted elements leaves a sorted list and after n insertions contains exactly the right elements. Thus, the algorithm must be correct.

How Efficient Insertion Sort is ?

Note that runtime changes with the permutations instance (even for a fixed size problem).

1. How does insertion sort do on sorted permutation ?
2. What about unsorted permutations ?

Analysis of Insertion Sort

Count the number of times each line of pseudo code will be executed.

Line	Insertion Sort (A, n)	# Inst (cos t)	# Exec (Time)
1.	For $j \leftarrow 2$ to length $[A]$ do	C_1	n
2.	Key $\leftarrow A\ [j]$	C_2	$n-1$
3.	/ * put $A\ [j]$ in to $A\ [1....j-1]$*/	C_3	/
4.	$i \leftarrow j-1$	C_4	$n-1$
5.	While $i > 0$ & $A\ [1] >$ key do	C_5	t_j
6.	$A[i+1] \leftarrow A[i]$	C_6	
7.	$i \leftarrow i+1$	C_7	
8.	$A[i+1] \leftarrow$ key	C_8	$n-1$

The 'for statement' is executed $(n-1)+1$ times, within for statement, 'key $\leftarrow A[j]$' is executed $(n-1)$ times.

Let $t_j \leftarrow 1 + 0$ the number of elements that have to be slide right to insert the jth item.

Step 5 is executed $t_2 + t_3 + ... + t_n$ times.

Step 6 is executed $t_2 - 1 + t_{3-1} + ... t_{n-1}$.

Now, add up the executed instructions for all pseudo code lines to get the run time of the algorithm :

$$T = C_1 . n + C_2 . (n-1) + C_4 . (n-1) + C_5 . \sum_{j=2}^{n} t_j + C_6 . \sum_{j=2}^{n} (t_j - 1)$$

$$+ C_7 . \sum_{j=2}^{n} (t_j - 1) + C_8 (n-1) \qquad ...(1.1)$$

What are the t_j's ? They depend on a particular input.

For best case : If it is already sorted, all t_j's are 1

$$t_j = 1$$

Hence, the best case time put $t_j = 1$ in eqn. (1.1) we get

$$T = C_1.n + C_2.(n-1) + C_4(n-1) + C_5 \sum_{j=2}^{n} 1 + 0 + 0 + C_8(n-1)$$

$$\Rightarrow \quad C_1 n + (C_2 + C_4 + C_5 + C_8)(n-1) = n + 1$$

where C and D are constants.

For worst case : If the input is sorted is decreasing order, we will have to slide all the alread[y] sorted element, So $t_j = j$ and step 5 is executed.

$$\Rightarrow \quad \sum_{j=2}^{n} j = \frac{n^2 + n}{2-1} = n^2 + n$$

1.7. FUNDAMENTAL OF ALGORITHM SOLVING

In computer science, developing an algorithm is an art or a skill. And we can have mastery [o]n algorithm development process only when we follow certain method. Before actual implementati[on] of the program, designing an algorithm is a very important step.

Supposes if we want to build a house we do not directly start constructing the house. Instead [w]e consult an architect, we put our ideas and suggestions, accordingly he draws a plan of the house a[nd] he discusses it with us. If we have some suggestion, the architect notes it down and makes t[he] necessary changes accordingly in the plan. This process continues till we are happy. Finally the bl[ue] print of house get ready. Once design process is over actual construction activity starts. Now [it] becomes very easy and systematic for construction of desired house. In this example, you will fi[nd] that all designing is just a paper work and at that instance if we want some changes to be done th[en] those can be easily carried out on the paper. After a satisfactory design the construction activit[y] starts. Now it becomes very easy and systematic for construction of desired house. In this examp[le] you will find that all designing is just a paper work and at that instance if we want some change[s to] be done then those can be easily carried out on the paper. After a satisfactory design the construct[ion] activities start. Same is a program development process. If we could follow same kind of appro[ach] while designing & analyzing the algorithm then we can have successful implementation for comp[lex] problems also. Let us list "what are the step, that need to be followed while designing & analys[ing] an algorithm ?"

These steps are :

1. Understanding the problem.
2. Decision making on
   (*a*) Capabilities of computational device.
   (*b*) Choice for either exact or approximate problem solving method.
   (*c*) Data structures
   (*d*) Algorithmic strategies.
3. Specification of algorithm.
4. Algorithmic verification.
5. Analysis of algorithm.

The 'n' space required for x [ ], one unit space for n, one unit for i and one unit for sum.

Algorithm 1.4. Add (x, n) ....

```
        //Problem Description : This is a recursive algorithm whic
        //computer addition of all the elements in an array x [ ]
        //Input : x [i] is of floating type, total number of elemen
        //in an array
        //Output : returns addition of n elements of an array
            return Add (x, n - 1)+ x [n]
```

The space requirement is

$$S(p) \geq 3(n + 1)$$

The internal stack used for resursion includes space for formal parameters, local variables an return address. The space required by each call to function Add requires atleast three words (spac for n values + space for return address + pointer to x [ ]). The depth of recursion in $n + 1$ (n time call to function and one return call). The recursion stack space will be $\geq 3(n + 1)$.

1.12.2. Time Complexity

The time complexity of an algorithm is the amount of computer time required by an algorith to run to completion.

It is difficult to compute the time complexity in terms of physically clocked time. For instanc in multiuser system, executing time depends on many factors such as :

(*i*) System load

(*ii*) Number of other programs running

(*iii*) Instruction set used

(*iv*) Speed of underlying hardware

The time complexity is therefore given in terms of frequency count.

Frequency count is a count denoting number of times of execution of statement.

1.12.3. Measuring an Input Size

It is observed that if the input size is longer, then usually algorithm runs for a longer time Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter. Sometimes to implement an algorithm we require prior knowledge of input size. For example, while performing multiplication of two matrices we should know order of these matrices Then only we can enter the elements of matrices. Sometimes the input size is taken as an approximate value. For example, in spell checking algorithms we can predict exact size of the input.

1.12.4. Measuring Running Time

We have already discussed that the time complexity is measured in terms of a unit called frequency count. The time which is measured for analyzing an algorithm is generally running time.

From an algorithm :

(*i*) We first identify the important operation (core logic) of an algorithm. This operation is called the basic operation.

(*ii*) It is not difficult to identify basic operation from an algorithm. Generally the operation which is more time consuming is a basic operation in the algorithm. Normally such basic operation is located in inner loop. For example in sorting algorithm the operation which is comparing the elements and then placing them at appropriate locations is a basic operation. The concept of basic operations can be well understood with the help of following example.

TABLE 1.1. Basic Operation From Input

Problem Statement	Input Size	Basic Operation
Searching a key element from the list of n elements.	List of n elements.	Comparison of key with every element of list.
Performing matrix multiplication.	The two matrices with order $n \times n$.	Actual multiplication of the elements in the matrices.
Computing GCD of two numbers	Two numbers.	Division.

(*i*) Then we compute total number of time taken by this basic operation. We can compute the running time of basic operation by following formula.

(*ii*) Using this formula the computing time can be obtained approximately.

$$T(n) = C_{op}\, C(n)$$

Running time of basic operation

Time taken by the basic operation to execute

Number of times the operation needs to be executed

Fig. 1.6.

1.12.5. Order of Growth

Measuring the performance of an algorithm in relation with the input size n is called order of growth. For example, the order of growth for varying input size of n is as given below.

TABLE 1.2. Order of Growth.

n	$\log n$	$n \log n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967.296

We will plot the graph for these values.

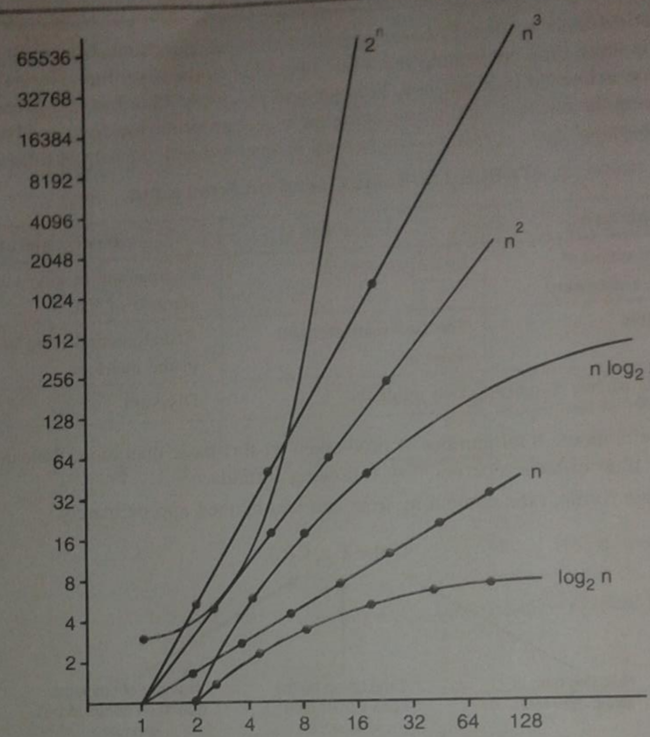


Fig. 1.7. Rate of growth of common computing time function.

From the above drawn graph it is clear that the logarithmic function is the slowest growing function. And the exponential function 2^n is fastest and grows rapidly with varying input size n. The exponential function gives huge values even for small input n. For instance for the value of $n = 16$ we get $2^{16} = 65536$.

1.12.6. Best Case, Worst Case and Average Case Analysis

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to to completion for a specific set of input then it is called worst case time complexity.

For example : While seaching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.

Consider the following algorithm :

Algorithm 1.5. Seq_search (x [0.... n − 1], key)

```
//Problem Description : This algorithm is for searching the
//key element from an array x[0...n - 1] sequentially.
//Input : An array x [0....n - 1] and search key
//Output : Returns the index of x where key value is present
for j ← 0 to n - 1 do
   if (x [i] = key) then
      return i
```

Best Case Time Complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element key is searched from the list of n elements. If the key element is present at first location in the list $\{x[0....n-1]\}$ then algorithm run for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{best} = 1$$

Worst Case Time Complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element key is searched from the list of n elements. If the key element is present at nth location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{worst} = n$$

The algorithm guarantees that for any instance of input which is of size n, the running time will not exceed $C_{worst}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average Case Time Complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and

P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at ith location. Hence probability of occuring first match is P/n for every ith element.

The probability of getting unsuccessful search is $(1 - P)$.

Now, we can find average case time complexity $C_{avg}(n)$ as

$C_{avg}(n)$ = Probability of succesful search (for elements 1 to n in the list)

+ Probability of unsuccessful search

$$C_{avg}(n) = \left[1.\frac{P}{n}+2.\frac{P}{n}+....+i.\frac{P}{n}\right]+n.(1-P)$$

n = There may be n elements at which chances of not getting element are possible. Hence $n.(1-P)$

$$= \frac{P}{n}[1+2+...+i...n]+n(1-P)$$

$$= \frac{P}{n}\frac{n(n+1)}{2}+n(1-P)$$

$$\boxed{C_{avg}(n) = \frac{P(n+1)}{2}+n(1+P)}$$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if $P = 0$ that means there is no successful search *i.e.*, we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation.

$$C_{avg}(n) = \frac{0(n+1)}{2}+n(1-0)$$

$$C_{avg}(n) = n$$

Thus the average case running time complexity becomes equal to n.

Suppose if $P = 1$ *i.e.* we get a successful search then

$$C_{avg}(n) = \frac{1(n+1)}{2}+n(1-1)$$

$$C_{avg}(n) = \frac{(n+1)}{2}$$

That means the algorithm scans about half of the elements from the list.

For calculating average case time complexity we have to consider probability of getting success of the operation. And any operation in the algorithm is heavily dependent on input elements. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

Amortized Analysis

The amortized time cannot be computed within one execution of an algorithm, rather over a sequence of operations we compute the time complexity. There may be a situation that a single operation is expensive but the time taken by sequence of n operations can be better than the worst case time. Hence

Amortized analysis means finding the average running time per operation over a worst case sequence of operations.

Thus the amortized analysis guaratnees the time per operation over the worst case performance. Amortized analysis assumes worst case input and typically does not allow random choices.

The average case analysis and amortized analysis are different. In average case analysis, we are averaging over all possible inputs whereas in amortized analysis we are averaging over a sequence of operations. The amortized analysis does not allow random selection of input whereas the average case time complexity is calculated by selecting random input from the list.

1.14. MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHM

First of all we will see the general plan for analyzing the efficiency of recursive algorith This plan tells us the steps to be followed while analyzing such algorithms.

1.14.1. General Plan for Analyzing Efficiency of Recursive Algorithms

1. Decide the input size based on parameter n.
2. Identify algorithm's basic operation (s).
3. Check how may times the basic operation is executed. Then find whether the execution basic operation depends upon the input size n. Determine worst, average, and best case input of size n. If the basic operation depends upon worst case, average case and best ca then that has to be analyzed separately.
4. Set up the recurrence relation with some initial condition and expressing the basic operatio
5. Solve the recurrence or atleast determine the order of growth. While solving the recurren we will use the forward and backward substitution method. And then correctness of formu can be proved with the help of mathematical induction method.

Let us analyze some recursive algorithms mathematically.

1.14.2. Examples Factorial (n)

1. *Computing Factorial of Some Number n.*

The factorial of some number can be obtained by performing repeated multiplication.

For instance : If $n = 5$ then

Step 1 : $n! = 5!$

Step 2 : $4! * 5$

Step 3 : $3! * 4 * 5$

Step 4 : $2! * 3 * 4 * 5$

Step 5 : $1! * 2 * 3 * 4 * 5$

Step 6 : $0! * 1 * 2 * 3 * 4 * 5$

Step 7 : $1 * 1 * 2 * 3 * 4 * 5$ as $0! = 1$

The above mentioned steps can be written in pseudocode form as :

Algorithm 1.9. Factorial (n)

```
//Problem Description : This algorithm computes n! using
//recursive function
//Input : A non negative integer n
//Output : returns the factorial value.
  if (n = 0)
    return 1
  else
    return factorial (n - 1) * n
```

Mathematical Analysis

Step 1 : The factorial algorithm works for input size n.

Step 2 : The basic operation in computing factorial is multiplication.

Step 3 : The recursive furnction call can be formulated as

$$F(n) = F(n-1) * n \qquad \text{where } n > 0$$

Then the basic operation multiplications is given as $M(n)$. And $M(n)$ is multiplication count to compute factorial (n).

$$M(n) = M(n-1) \quad + \quad 1$$

These multplications are required to compute factorial $(n-1)$.

To multiply factorial $(n-1)$ by n.

Step 4 : In step 3 the recurrence relation is obtained.

$$M(n) = M(n-1) + 1$$

Now we will solve recurrence using

Forward Substitution

$$M(1) = M(0) + 1$$

$$M(2) = M(1) + 1 = 1 + 1 = 2$$

$$M(3) = M(2) + 1 = 2 + 1 = 3$$

Backward Substitution

$$M(n) = \underbrace{M(n-1)}_{\downarrow} + 1$$

$$= \left[\underbrace{M(n-2)+1}_{\downarrow}\right] + 1 = M(n-2) + 2$$

$$= [M(n-3) + 1] + 1 + 1 = M(n-3) + 3$$

From the substitution methods we can establish a general formula as :

$$M(n) = M(n-i) + i$$

Now let us prove correctness of this formula using mathematical induction as follows :

Prove $M(n) = n$ by using mathematical induction.

Basis : Let $n = 0$ then

$$M(n) = 0$$

i.e. $M(0) = 0 = n$

Induction : If we assume $M(n-1) = n - 1$ then

$$M(n) = M(n-1) + 1$$

$$= n - 1 + 1$$

$$= n$$

i.e. $M(n) = n$

Thus the time complexity of factorial function is $\Theta(n)$.

1.14.3. Example : Fibonacci Numbers

This series is identified by European Mathematician Leonardo Fibonacci in 1202. The Fibonacci numbers are given by following series.

0, 1, 1, 2, 3, 5, 8

This series can be obtained by following formula

$$f(n) = f(n-1) + f(n-2) \quad \text{where } n > 1$$

i.e. previous two numbers are added to generate current number.

There are two initial conditions,

$$\text{Fib}(0) = 0$$
$$\text{Fib}(1) = 1$$

This recursive definition of Fibonacci series can be shown diagrammatically as,

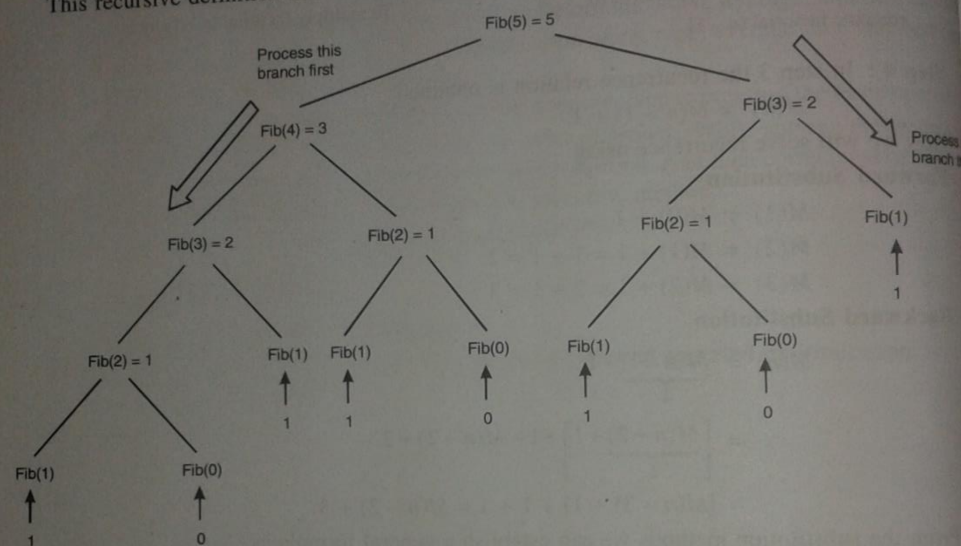


Fig. 1.8. Recursive calls made to obtain nth Fibonacci number.

Algorithm 1.10. To obtain nth Fibonacci number

```
Algorithm Fib (n)
//Problem Description : The algorithm finds the nth
//Fibonacci number
//Input : The number n for which the value of Fibonacci serie
//is to be obtained.
//Output : The nth Fibonacci number is returned.
   if (n <- 1) then
      return n
      x ← Fib (n - 1)
      y ← Fib (n - 2)
      return (x + y)
```

The program for this algorithm is given below.

```
/***********************************************************************
    Program to find nth Fibonacci number
************************************************************************/
    # include <stdio.h>
    # include <conio.h>
    void main ( )
    {
       int n;
       int fib (int n) ;
       clrscr ( ) ;
       printf ("\n Enter the value of n for Fibonacci number");
       scanf ("%d", & n);
       printf ("\n The value at %d th index in Fibonacci series is", n);
       printf ("%d", fib(n));
       getch ( );
    }
    int fib (int n)
    {
       int x, y ;
       if (n <= 1)
       return n;
       x = fib (n - 1);
       y = fib (n - 2);
       return (x + y);
    }
```

Output

```
Enter the value of n for Fibonacci number5
The value at 5th index in Fibonacci series is 5.
```

1.14.3.1. Explicit Formula for nth Fibonacci Number

The algorithm for Fibonacci series always requries n number to compute nth value. Hence input size of this algorithm is n. And the basic operation in this algorithm is to perform addition of previous two numbers. But obtaining formula for nth Fibonacci number with recurrence relation using forward and backward substitution method is complex. Hence we will use homogeneous second order linear recurrence with constant coefficients to obtain nth Fibonacci number. The linear second order recurrence with constant coefficient is a special class of recurrence in which solution is neither obtained by forward or backward substitution. The recurrences are of the type

$$ax(n) + bx(n-1) + cx(n-2) = F(n) \quad ...(1$$

where a, b and c are real numbers and $a \neq 0$. This type of recurrence is of second order because $x(n)$ and $x(n-2)$ are two positions apart in the given sequence. This sequence is called linear because L.H.S. contains linear combinations of unknown terms. If $F(n) = 0$, then sequence is called homogeneous otherwise it is called inhomogeneous. And as a, b, c are some constant terms appearing in the sequence this sequence is called with constant coefficients. If the sequence is homogeneous, then

$$ax(n) + bx(n-1) + cx(n-2) = 0 \qquad \text{...(2)}$$

The solution to this equation (2) can be obtained by using quadratic equation.

$$ar^2 + br + c = 0 \qquad \text{...(3)}$$

This equation (3) is called **characteristic equation**.

Now to obtain formula for nth Fibonacci number we will rewrite equation (1) as,

$$F(n) = F(n-1) + F(n-2)$$

i.e.,
$$F(n) - F(n-1) - F(n-2) = 0$$

The characteristic equation for above equation is,

$$r^2 - r - 1 = 0$$

We can solve the quadratic equation by finding its roots.

The formula for finding roots of quadratic equation is $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

1.15. SEARCHING ALGORITHMS

	Worst Case	Average Case	Best Case
Linear Search	$0(n)$	$0(n)$	$0(1)$
Binary Search	$0(\log_2 n)$	$0(\log_2 n)$	$0(1)$

NOTE

Binary search is better than linear search for worst case and average case conditions. But for binary search the elements should be first arranged in ascending or decending order. This involves an extra effort. On the other hand, for a sorted array, if linear search algorithm is used, there may not be much improvement in the performance.

1.16. SORTING ALGORITHM

	Worst Case	Average Case	Best Case
Bubble	$0(n^2)$	$0(n^2)$	$0(n)$
Insertion	$0(n^2)$	$0(n^2)$	$0(n)$
Selection	$0(n^2)$	$0(n^2)$	$0(n^2)$
Quicksort	$0(n^2)$	$0(n \log n)$	$0\ (n \log n)$

EXERCISES

1. What is an algorithm ? Explain various criteria used for judging an algorithm.
2. How do you say one algorithm is better than another algorithm for solving the same problem ?
3. Differentiate between profiling and debugging.
4. Differentiate between priori analysis and posteriori analysis.

QUESTIONS WITH ANSWERS

Q. 1. What is an algorithm ? Explain various criteria used for judging an algorithm.

Ans. An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. In addition, the algorithm must satisfy the following criteria (norms).

(*i*) **Input :** No input or more data items are supplied.

(*ii*) **Output :** At least one data item is produced.

(*iii*) **Definiteness :** Each instruction is clear and unambiguous.

(*iv*) **Finiteness :** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(*v*) **Effectiveness :** Every instruction must be basic so that a person using pencil and paper can carry it out, in principle. It is not enough that each operation is definite as in criterion (3); it also must be feasible.

Q. 2. How do you say one algorithm is better than another algorithm for solving the same problem ?

Ans. The optimal computing time and storage requirements of an algorithm decide the better algorithm. Every algorithm must use some of a computer's resources to complete its task. The resources most relevant to efficiency are central processor time (CPU time) and internal memory (RAM). In other words, the two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity.

Q. 3. Differentiate between profiling and debugging.

Ans. Testing a program consists of two phases : debugging and profiling (or performance measurement). Debugging is the process of executing programs on sample data sets to find whether wrong results occur and, if so, to correct them. Debugging can only point to the presence of errors. More than one programmer can develop programs for the same problem and verify the output with different input data sets.

Profiling is the process of executing a correct program on data sets and measuring its time and space. These timing figures are used to confirm the previous analysis and point out logical errors. This way, we can optimize the performance of algorithms.

Q. 4. Differentiate between priori analysis and posteriori analysis.

Ans. In order to choose between data structures and algorithms, the efficiency of competing data structures (in terms of space required) and competing algorithms (in terms of time used) must be compared. The best test of a data structure or algorithm is to measure its efficiency. This is called a posteriori (after the fact) analysis. Since the choice of a data structure or algorithm is generally made before a program is written, a priori (before the fact) analysis is necessary.

The priori estimates refer to performance analysis and the posteriori analysis to performance measurement. A frequency count of statements executed is the most direct form of a priori analysis of the time used by an algorithm. A frequency count of data structure elements used is the most direct form of a priori analysis of data structures.

Q. 5. Find complexity of the following loop ?

```
conside the following loop
    for i → 1 to m
        {
            P(i)
        }
```

Ans. In this type of algorithm, the premitive or basic statement is $P(i)$. If we consider that the execution time of $P(i)$ is to then the total time taken by the above kind of algorithm is $\sum_{i=1}^{m} tl$.

Note : Generally we take the execution time of premtive statements as units.

$$\text{So total time } = \sum_{i=1}^{m} \Theta(1) = \Theta \sum_{i=1}^{m} 1 = \Theta(m)$$

So the time complexity is $\Theta(m)$.

Q. 6. Find complexity of the following nested loop ?

```
for i ← 1 to m
  { for j ← 1 to m
      {
          P(ij)
      }
  }
```

Ans. Time complexity = $\sum_{i=1}^{m} \sum_{i=1}^{m} \Theta(1) = \Theta(m^2)$

Q. 7. Consider the following loop and calculate the total computation time.

```
for i ← 2 to m - 1
  {
     For j ← 3 to i
     {
         sum ← sum + A[i] [j]
     }
  }
```

Ans. Total computing time $= \sum_{i=2}^{m-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{m-1} \sum_{j=3}^{i} \Theta(1) = \sum_{i=2}^{m-1} \Theta(i)$

$$= \Theta\left(\sum_{i=2}^{m-1} (i)\right) = \Theta\left(\frac{m^2}{2} + \Theta(m)\right) = \Theta(m^2)$$

OBJECTIVE TYPE QUESTIONS

1. The step-by-step description of a process is called as
   (a) Algorithm (b) Pseudocode
   (c) Flowchart (d) Program

2. Pseudocode is
   (*a*) Refined version of program
   (*b*) Language independent code
   (*c*) Outcome of compilation process
   (*d*) Code developed using the syntax of a specific language
3. Experimental methods are used in
   (*a*) Performance analysis
   (*b*) Performance measurements
   (*c*) Efficiency
   (*d*) Performance analysis and performance measurements
4. The components of time complexity are
   (*a*) The compiler used
   (*b*) The target compute
   (*c*) The computer options
   (*d*) The compiler, the computer options, the target computer
5. The approaches used to determine the performance of a program are
   (*a*) Analytical
   (*b*) Analytical and experimental
   (*c*) Analytical the theoretical
   (*d*) Experimental

ANSWERS

1. (*a*) 2. (*b*) 3. (*d*) 4. (*d*) 5. (*b*)

Chapter 2

Growth of Functions

INSIDE THIS CHAPTER

Asymptotic Notations is used to describe the running time of an algorithm. This shows the order of growth of function. We can map the time taken by an algorithm in terms of mathematical function. For example an algorithm taking n^2 *microseconds* can be thought of "of the order of n^2", as unit is constant. Asymptotic notations are mathematical way of representating "order of". We have many asymptotic notations like θ, O Ω, ω, o described in coming sections.

2.1. ASYMPTOTIC NOTATIONS

There exists an algorithm for every given problem in the world, and this algorithm should always be analysed by calculating its total time complexity *i.e.,* total time required by the algorithm to execute itself. This time complexity is always calculated and it depends on the following :

(*a*) Total number of inputs in the algorithm.

(*b*) Total number of comparisons

and it always generated in mathematical form. But if we want to represent the time complexity in term of algorithm, then some notations are used called asymptotic notations, where asymptotically indicate that the time is exactly equal to given time *i.e.,* it is tightly bounded.

Asymptotic notation are further catagorized into the following five notations depending on the total time complexity of an algorithm.

(*a*) Big Oh ('O')

(*b*) omega ('Ω')

(*c*) theta ('θ')

(*d*) little Oh ('o')

(*e*) little omega ('ω')

2.1.1. Big Oh Notations

This notation always generates the worst time or the maximum time required by a particular algorithm.

$\boxed{f(n) = O(g(n)), \text{ there exists some constants } C > 0 \text{ for which } 0 \le f(n) \le C.g(n) \ \forall \ n \ge n_0.}$

where $f(n)$ and $g(n)$ are the function of time, $f(n)$ is the actual time while $g(n)$ is the maximum time.

'C' is any constant and n_0 are the minimum number of elements such that time function $f(n)$ should have lesser or equal value compared to $C.g(n)$

Big Oh ('O') notation is also known as asymptotically upper bound and can be represented graphically using Fig. 2.1.

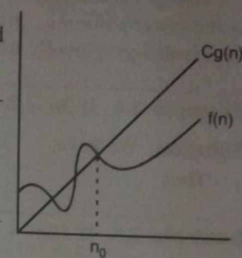


Fig. 2.1. Asymptotic function graph : $f(n) = O(g(n))$

Example 2.1. If $3n + 2 = O(n)$, find $g(n)$.

Solution. We have $3n + 2 \le 4n \ \forall \ n \ge 2$

Thus $g(n) = n, C = 4, n_0 = 2.$

Example 2.2. If $10n^2 + 4n + 2 = O(n^2)$, find $g(n)$.

Solution. We have $f(n) = 10n^2 + 4n + 2 \le 11n^2 \quad \forall \ n \ge 5$

Thus $g(n) = n^2, C = 11, n_0 = 5.$

Example 2.3. If $f(n) = 6.2^n + n^2 = O(2^n)$, find $g(n)$.

Solution. We have $f(n) \le 7.2^n \ \ \forall \ n \ge 4$

Thus $g(n) = 2^n, C = 7, n_0 = 4$

2.1.1.1. Running Time of an Algorithm

The running time,in general can be in the following order

$$O(1) < O(\log_2 n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Function Values

lg n	*n*	*n lg n*	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	5	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65, 536
5	32	160	1024	32768	4,294,967,296

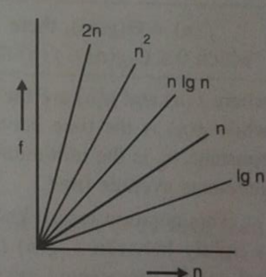


Fig. 2.2. Shows approximate plot of functions

2^n grows very rapidly. See Fig. 2.2.

NOTE

In these type of problems choose n as power of 2.

2.1.2. Omega Notation

This notation always generates the best time or the minimum time required by any algorithm.

$\boxed{f(n) = \Omega(g(n)), \text{there exists some constants } C > 0 \text{ for which } 0 \le C.g(n) \le f(n) \ \ \forall \ n \ge n_0}$

where $f(n)$ and $g(n)$ are the function of time, $f(n)$ is the actual time while $g(n)$ is the minimum time, 'C' is any constant and n_0 are the minimum number of elements such that time function $f(n)$ should have greater or equal values compared to $C.g(n)$.

Omega notation (Ω) is also known as asymptotically lower bound as the lower value *i.e.*, $g(n)$ is fixed at some lower time and can be represented graphically using Fig. 2.3.

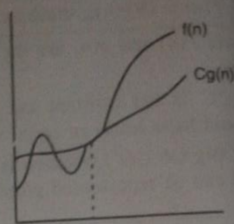


Fig. 2.3. Asymptotic function graph : $f(n) = \Omega(g(n))$

Example 2.4. If $3n + 2 = \Omega(n)$, find $g(n)$.

Solution. We have $3n + 2 \geq 3n \ \forall\ n \geq 1$

Thus $g(n) = n,\ C = 3,\ n_0 = 1.$

Example 2.5. If $10n^2 + 4n + 2 = \Omega(n^2)$, find $g(n)$.

Solution. We have $10n^2 + 4n + 2 \geq 10n^2 \ \forall\ n \geq 1$

Thus $g(n) = n^2,\ C = 10,\ n_0 = 1.$

Example 2.6. If $f(n) = 6.2^n + n^2 = \Omega(2n)$, find $g(n)$.

Solution. We have $f(n) \geq 6.2^n \ \forall\ n \geq 1$

Thus $g(n) = 2^n,\ C = 6,\ n_0 = 1.$

2.1.3. Theta Notation

Theta notation ('Θ') always generates the average time or the time between maximum *i.e.*, big oh ('O') and minimum *i.e.*, omega ('Ω') required by any algorithm.

> $f(n) = \Theta(g(n))$, there are two constants $C_1 > 0$ and $C_2 > 0$ for which $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n) \ \forall\ n \geq n_0$

where $f(n)$ and $g(n)$ are the functions of time, $F(n)$ is the actual time while $g(n)$ is the time between minimum and maximum 'C' is any constant. n_0 is the minimum number of elements for which theta (Θ) generates average time.

This notation is also known as asymptotically tight bound as $f(n)$ is tightly between $C_1 g(n)$ *i.e.*, minimum and $C_2 g(n)$ *i.e.*, maximum and can be represented graphically using Fig. 2.4.

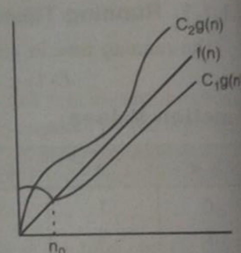


Fig. 2.4. Asymptotic function graph : $f(n) = \Theta(g(n))$

Example 2.7. $f(n) = \frac{1}{2}n^2 - 3n$, find Θ notation.

Solution. Since $\Theta(g(n))$ is a set and $f(n)$ is its member by definition. We can assume $g(n) = n^2$.

$$C_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq C_2 n^2$$

$$\Rightarrow \quad C_1 \leq \frac{1}{2} - \frac{3}{n} \leq C_2$$

$$C_2 \geq \frac{1}{2} - \frac{3}{n}$$

Since C_2 is +ve constant, C_2 have maximum value $\frac{1}{2}$ as –ve term is subtracted.

$$\therefore \quad C_2 \geq \frac{1}{2}$$

and $$C_1 \leq \frac{1}{2} - \frac{3}{n} \quad \forall\, n = 7, C_1 \text{ is +ve so,}$$

$$C_1 = \frac{1}{2} - \frac{3}{n} = \frac{1}{14}$$

$$\therefore \quad C_1 \leq \frac{1}{14}$$

Hence, lowest value of 'n' for which $f(n)$ lies between $C_1 g(n)$ and $C_2 g(n)$ is $n_0 = 7$.
Hence order is $\Theta(n^2)$.

Example 2.8. Find Θ-notation for $\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$.

Solution. Let
$$f(n) = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$
$$= \left(\frac{1}{3} + \frac{1}{2n} + \frac{1}{6n^2}\right)n^3$$
$$\leq \left(\frac{1}{3} + \frac{1}{2} + \frac{1}{6}\right)n^3 \ ; \ n \geq 1$$
$$\leq C_2 n^3$$

Let
$$g(n) = n^3$$
$$f(n) \leq C_2 g(n)$$
$$\therefore \quad f(n) = \Theta(n^3)$$

Theorem 2.1. If $f(n) = a_0 + a_1 n + \ldots + a_m n^m$ is any polynomial of degree m or less, then $f(n) = \theta(n^m)$

Proof. We have
$$|f(n)| \leq |a_0| + |a_1| n + |a_2| n^2 + \ldots + |a_m| n^m$$
$$= \left(\frac{|a_0|}{n^m} + \frac{|a_1|}{n^{m-1}} + \ldots + |a_m|\right) n^m$$
$$= (|a_0| + |a_1| + \ldots + |a_m|) n^m, \ n \geq 1$$

When $n = 1$,
$$C_2 = |a_0| + |a_1| + \ldots |a_m|$$
$$\therefore \quad f(n) \leq C_2 g(n) \text{ where } g(n) = n^m.$$

2.4. BIG-OH RULES

(a) If $f(n)$ is a polynomial of degree K, then $f(n)$ is $O(n^K)$, *i.e.*, drop lower-order terms and constant factors.

(b) Use the smallest possible class of function-say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$".

(c) Use the simplest expression of the class-say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$".

2.4.1. Running Time of Algorithm Using O-Notation

In computing, we categorize the algorithms according to their asymptotic run-time functions. The run-time behaviour of most algorithms falls into one of only about seven primary categories $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$. These are known as complexity classes or asymptotic growth classes.

TABLE 2.1. Basic Asymptotic Efficiency Classes.

Name of Efficiency Class	Order of Growth	Description	Example
Constant	1	As input size grows then we get larger running time.	Scanning array elements.
Logarithmic	$\log n$	When we get logarithmic running time then it is sure that the algorithm does not consider all its input rather the problem is divided into smaller parts on each iteration.	Performing binary search operation.
Linear	n	The running time of algorithm depends on the input size n.	Performing sequential search operation.
$n \log n$	$n \log n$	Some instance of input is considered for the list of size n.	Sorting the elements using merge sort or quick sort.
Quadratic	n^2	When the algorithm has two nested loops then this type of efficiency occurs.	Scanning matrix elements.
Cubic	n^3	When the algorithm has three nested loops then this type of efficiency occurs.	Performing matrix multiplication.
Exponential	2^n	When the algorithm has very faster rate of growth then this type of efficiency occurs.	Generating all subsets of n elements.
Factorial	$n!$	When an algorithm is computing all the permutations then this type of efficiency occurs.	Generating all permultations.

SOLVED EXAMPLES

Example 2.17. $2n + 10$ **is** $O(n)$

Solution. By O-notation, we need to find a real constant $C > 0$ and an integer constant $n_0 \geq 1$, such that

$$2n + 10 \leq Cn$$
$$2n + 10 \leq Cn$$
$$(C-2)n \geq 10$$
$$n \geq \frac{10}{(C-2)}$$

A possible choice is $C = 3$ and $n_0 = 10$. This is one-to-many choices.

Example 2.18. n^2 **is not** $O(n)$

Solution.
$$n^2 \leq Cn$$
$$n \leq C \text{ (after dividing by } n \text{ on both sides)}$$

This inequality cannot be satisfied since 'C' must be a constant.

Example 2.19. $(n + 1)^2 = n^2 + 2n + 1$ **is** $O(n^2)$.

Solution.
$$(n + 1)^2 \leq Cn^2$$

This need $C > 0$ and $n_0 \geq 1$ such that

$$(n + 1)^2 \leq Cn^2 \text{ for } n \geq n_0$$

This is true for $C = 4$, $n_0 = 1$ *i.e.*,

$$3(21)^3 + 20(21)^2 + 5 \leq 4(21)^3$$
$$36608 \leq 37044$$

Example 2.20. $\log n + \log \log n$ **is** $O(\log n)$.

Solution. This needs $C > 0$ and $n_0 \geq 1$ such that

$$\log n + \log \log n \leq C \log n \ \forall\ n \geq n_0.$$

This is true for $C = 4$, $n_0 = 2$ *i.e.*,

$$\log (2) + \log \log (2) \leq 4(\log (2))$$
$$1 + \log (1) \leq 4(1) \qquad [\text{as } \log_2 2 = 1]$$
$$1 + 0 \leq 4$$

Example 2.21. $f(n) = 3n^2 + 7n - 5$.

Solution. For large values of n, $7n - 5$ is insignificant compared to $3n^2$, so we would except that $f(n)$ is $\Theta(3n^2)$. In fact, since constant multiplies are not significant, we expect that $f(n)$ is $\Theta(n^2)$. We could note that for $n \geq 1$, $7n - 5 \geq 0$ and so $3n^2 + 7n - 5 \geq 3n^2$. And we could note that for $n \geq 7$.

$$3n^2 + 7n - 5 \leq 3n^2 + 7n \leq 3n^2 + n^2 \leq 4n^2$$
$$3(7^2) + 7(7) - 5 \leq 3(7)^2 + 7(7) \leq 3(7)^2 + (7)^2 \leq 4(7)^2$$
$$191 \leq 196 \leq 196 \leq 196$$

So, we have taking $g(n) = n^2$, that $3g(n) \le f(n) \le 4g(n)\ \forall\ n \ge 7$.

Taking $n_0 = 7$, $C_1 = 3$ and $C_2 = 4$ in the definition, we have proved that $f(n)$ is $\Theta(n^2)$.

Example 2.22. Prove or disprove

(*a*) $2^{n+1} = O(2^n)$

(*b*) $2^{2n} \ne O(2^n)$.

Solution. (*a*)

$$f(n) = 2^{n+1}$$

$$2^{n+1} \le C.g(n) \qquad \text{where } g(n) = 2^n$$

$$\therefore \quad 2^{n+1} \le C.2^n$$

$$2^{n+1} \le 2.2^n \qquad \text{where } C = 2$$

if $C \ge 2$ then $2n + 1 = O(2^n)$ **Proved.**

(*b*)

$$f(n) = 2^{2n}$$

$$2^{2n} \le C.g(n) \qquad \text{where } g(n) = 2^n$$

$$2^{2n} \le C.2^n$$

$$\therefore \quad 2^{2n} \le 2^n.2^n \qquad \text{where } C = 2^n$$

if $C \ge 2^n$ then $2^{2n} = O(2^n)$

which is not possible as C depends on the value of 'n' which is not a constant value, hence $2^{2n} \ne O(2^n)$.

Example 2.23. Prove the following :

(*a*) $n! = O(n^n)$

(*b*) $\log (n!) = O(n \log n)$

(*c*) $1^k + 2^k + ... + n^k = O(n^{k+1})$

(*d*) $1 + 2 + ... + n = O(n^2)$.

(*e*) $(n + a)^b = O(n^b)$

Solution. (*a*)

$$f(n) = n!$$

$$n! = n(n-1)(n-2) .... 1$$

$$n! \le C.g(n) \text{ as big-oh notation.}$$

$$n(n-1)(n-2) .... 1 \le C.g(n)$$

$$n(n-1)(n-2) .... 1 \le n\ .\ n\ .\ n.\ ....\ n \text{ times}$$

$$n! \le n^n$$

Hence $n! = O(n^n)$, $C = 1$, $n_0 = 1$ **Proved.**

(*b*)

$$f(n) = \log (n!)$$

$$n! = n(n-1)(n-2) ... 1$$

$$\log (n!) \le C.g(n) \text{ as big-oh notation}$$

$$\log [n(n-1)(n-2) .... 1] \le \log [n\ .\ n.\ n.\ ....\ n \text{ times}]$$

$$\log (n!) \le \log (n^n)$$

$$\log (n!) \le n \log n$$

Hence $\log(n!) = O(n \log n)$, $C = 1$, $n_0 = 1$ **Proved.**

(c)

$$\begin{aligned} f(n) &= 1^k + 2^k + ... + n^k \\ f(n) &\le C.g(n) \text{ as big-oh notation} \\ 1^k + 2^k + ... + n^k &\le n^k + n^k + ... + n^k \\ &\le n.n^k \\ &\le n^{k+1} \end{aligned}$$

Hence $f(n) = O(n^{k+1})$, $C = 1, n_0 = 1$ **Proved.**

(d)

$$\begin{aligned} f(n) &= 1 + 2 + ... + n \\ f(n) &\le C.g(n) \qquad \text{as big-oh notation} \\ 1 + 2 + ... + n &\le n + n + ... + n \\ &\le n\,.\,n \\ &\le n^2 \end{aligned}$$

Hence $f(n) = O(n^2)$, $C = 1$, $n_0 = 1$ **Proved.**

(e)

$$\begin{aligned} f(n) &= (n + 1)b \\ f(n) &= (g(n))^b \text{ suppose } g(n) = n + a \\ g(n) &= n + a \\ g(n) &\le C.g(n) \text{ as big-oh notation.} \\ n + a &\le n + n \ \ \forall\, n_0 \ge |a| \\ n + a &\le 2n \qquad ...(1) \end{aligned}$$

Now

$$\begin{aligned} f(n) &\le C.g(n) \\ (n + a)^b &\le (2n)^b \qquad \text{using (1)} \\ &\le 2^b n^b \end{aligned}$$

Hence $(n + a)^b = O(n^b)$, $C = 2b$, $n_0 = |a|$.

Example 2.24. Prove that $\frac{n^2}{2} = \omega(n)$.

Solution. We know that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n^2}{2 \times n} = \frac{n}{2} = \infty$$

Thus, we say $\qquad \frac{n^2}{2} = \omega(n)$

Example 2.25. $C_1 n^2 + C_2 n \le C_3 n$

Solution. If $C_1 = 1$, $C_2 = 2$, $C_3 = 1000$

$$\Rightarrow \qquad n^2 + 2n \le 1000n.$$

$$n^2 \le 998n \quad n \le 998$$

No matter what the value C_1, C_2, C_3 takes, there will be n beyond which the algorithm with complexity $C_3 n$ will be faster than the one with $C_1 n^2 + C_2 n$. This value on is called the break-even

point. If break even point is zero, then the algorithm with complexity C_3n will always be faster (or at least as fast)

Example 2.26. **(i) Prove** $3n + 2 = o(n^2)$.

Solution.

$$\operatorname*{Lt}_{n\to\infty} \frac{3n+2}{n^2} = \operatorname*{Lt}_{n\to\infty} \frac{3}{n} + \frac{2}{n^2} = 0.$$

Thus, $o(n^2)$ holds.

Example 2.27. **Prove that** $lg\,(n\,!) = O(n\ lg\ n)$

Solution.

$$n\,! = n(n-1)\,(n-2)\,...\,3.2.1.$$

$$\therefore \quad \geq \left(\frac{n}{2}\right)^{n/2}$$

Taking log of both sides, (To the base 2)

$$lg\ n! \geq \frac{n}{2} lg \frac{n}{2}$$

$$= \frac{n}{2}(\lg n - \lg_2 2)$$

$$= \frac{n}{2}(\lg n - 1) = \frac{n}{2}\log n - \frac{n}{2}$$

$$\leq n \lg n.$$

$$= O(n \lg n)$$

Example 2.28. **Prove** $2^{n+1} = O(2^n)$.

Solution. By formula

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = 2$$

$$\therefore \quad f(n) = O(2^n)$$

Example 2.29. $(n + a)^b = \theta(n^b)$, $b > 0$, a **and** b **are real constants.**

Solution.

$$f(n) = n^b\left(1+\frac{a}{n}\right)^b$$

$$g(n) = n^b$$

$$\operatorname*{Lt}_{n\to\infty} \frac{n^b\left(1+\frac{a}{n}\right)^b}{n^b} = 1 = \text{constant as}$$

$$\boxed{\begin{array}{l}\text{if } \operatorname*{Lt}_{n\to\infty} \dfrac{f(n)}{g(n)} = \text{constant} \\ \text{then } f(n) = O(g(n))\end{array}}$$

$\therefore\ \theta(n^b)$ holds.

Example 2.30. **Prove $n\,! = o(n^n)$.**

Solution. We have

$$n\,! = n(n-1)\ \dots\ 3.2.1.$$

$$= n^n\left(1-\frac{1}{n}\right)\dots\frac{3}{n}.\frac{2}{n}.\frac{1}{n}.$$

$$\underset{n\to\infty}{\text{Lt}}\frac{f(n)}{g(n)} = \frac{n^n\left(1-\frac{1}{n}\right)\dots\frac{3}{n}.\frac{2}{n}.\frac{1}{n}}{\underset{n\to\infty}{\text{Lt}}\ n^n} = 0$$

$$\therefore \qquad n! = o(n^n)$$

Example 2.31. **Prove $lg\ n! = \theta(n\ lg\ n)$.**

Solution. We have $lg\ n\,! = n\ lg\ (n/e)$ from stirlings' approximation

$$= n\ lgn - n\ lg_2\ e$$

$$\underset{n\to\infty}{\text{Lt}}\frac{f(n)}{g(n)} = \frac{n\lg n - n\lg e}{\underset{n\to\infty}{\text{Lt}}\ n\lg n}$$

$$= 1 - \frac{\lg e}{\lg n} = 1$$

$$\therefore \qquad \lg n! = \theta(n\ lg\ n)$$

$\underset{n\to\infty}{\text{Lt}}$ (as $\log\infty = \infty$)

Example 2.32. **Prove $n\,! = \omega(2^n)$.**

Solution. We have $\underset{n\to\infty}{\text{Lt}}\dfrac{f(n)}{g(n)} = \infty$ for $f(n) = \omega(g(n))$

$\Rightarrow$ We can write,

$$n! = n^n$$

Applying $\underset{n\to\infty}{\text{Lt}}\dfrac{f(n)}{g(n)} = \dfrac{n^n}{2^n} = \left(\dfrac{n}{2}\right)^n$

Using L' Hospitals Rule

$$\underset{n\to\infty}{\text{Lt}}\frac{f(n)}{g(n)} = \frac{n^n}{1} = \frac{nn^n - 1}{0} = \infty$$

Thus $\qquad n\,! = \omega(2^n)$

NOTE

Here we can't apply F2.1 as in the formula $\underset{n\to\infty}{\text{Lt}}\dfrac{n^b}{a^n} = 0$, where b and a must be constant, but here, in $\dfrac{n^n}{2^n}$, b is not constant.

Hence it holds.

Example 2.33. Prove that $f(n) = lgn$ is $O(n^{\alpha})$, for any $\alpha > 0$

Solution. By definition, $f \varepsilon O(n)$ if

$$\operatorname{Lt}_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\frac{f(n)}{g(n)} = \frac{\lg n}{n^{\alpha}} = \operatorname{Lt}_{n\to\infty} \frac{\ln n}{\ln 2 \cdot n^{\alpha}}$$

$$\frac{f'(n)}{g'(n)} = \frac{\frac{1}{n}}{\ln 2 \cdot \alpha \cdot n^{\alpha-1}} = \frac{n}{n\,\alpha \ln 2\, n^{\alpha}} = \frac{1}{\alpha \ln 2 \cdot n^{\alpha}}$$

By L'Hospital Rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \operatorname{Lt}_{n\to\infty} \frac{f'(n)}{g'(n)} = \operatorname{Lt}_{n\to\infty} \frac{1}{(\alpha \ln 2) \cdot n^{\alpha}} = \frac{1}{\alpha \ln 2 \cdot n^{\alpha}} = 0$$

$\therefore$ $f \varepsilon O(g)$ proved.

Example 2.34. Prove that $\sum_{i=1}^{n} \log(i)$ is $\theta(n \log n)$

Solution. Let

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} \log(i) \\ &= \log 1 + \log 2 + \log 3 + \log 4 + \ldots + \log n \\ &= \log 2 \,.\, 3 \,.\, 4 \ldots n, \text{ as } \log 1 = 0. \\ &= \log (n-1)\,! \\ &\le \log n\,! \end{aligned}$$

But $\log n! = \theta(n \lg n)$

$\therefore$ **$f(n)\ \varepsilon\ \theta(n\ lg\ n)$ proved**

Note : log 1 is zero to any base.

Example 2.35. ***Compute x^n, write an algorithm and improve its running time.***

Solution.

```
Power ← x
For i ← 1 to n - 1 do
Power ← Power × x.
```

This algorithm takes $\theta(n)$ time.

We can modify this algo to reduce the runtime, Let $2^K = n$, s.t. $K = \lg n$, then,

```
Power ← x.
   for i ← 1 to k do
   Power = Power^2
```

This algo. executes in $\theta(\log n)$ time this is significant improvement over the previous algori

Example 2.36. What does all this mean?

$3n^2 - 100n + 6 = O(n^2)$ because $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ because $01n^3 > 3n^2 - 100n + 6$
$3n^2 - 100n + 6 \neq O(n)$ because $c.n < 3n^2$ when $n > c$
$3n^2 - 100n + 6 = \Omega(n^2)$ because $2.99n^2 < 3n^2 - 100n + 6$
$3n^2 - 100n + 6 \neq \Omega(n^3)$ because $3n^2 - 100n + 6 < n^3$
$3n^2 - 100n + 6 = \Omega(n)$ because $10^{10^{10}} n < 3n^2 - 100 + 6$
$3n^2 - 100n + 6 = \theta(n^2)$ because O and Ω
$3n^2 - 100n + 6 \neq \theta(n^3)$ because O only
$3n^2 - 100n + 6 \neq \theta(n)$ because Ω only

Note that

$$\log (n^{473} + n^2 + n + 96) = O(\log n)$$
$$n^{473} + n^2 + n + 96 = O(n^{473}), \text{ and}$$

Since $\log n^{473} = 473 \cdot \log n$

Any exponential dominates every polynomial. This is why we will seek to avoid exponential time algorithms.

Example 2.37. For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t assuming that the algorithm to solve the problem takes $f(n)$ microseconds. Recall that $\log n$ denotes the logarithm in base 2 of n. Some entries have already been completed to get you started.

Solution. The numbers in the first row are quite large. The table below calculates it approximately in powers of 10. People might also choose to use powers of 2. Being close to the answer is enough for the big numbers (within a few factors of 10 from the answers shown).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$2^{10^6} \approx 10^{300000}$	$2^{3.6 \times 10^9} \approx 10^{10^9}$	$2^{2.6} \times 10^{12} \approx 10^{0.8} \times 10^{12}$	$2^{3.1 \times 10^{15}} \approx 10^{10^{15}}$
$\sqrt{n}$	$\approx 10^{12}$	$\approx 1.3 \times 10^{19}$	$\approx 6.8 \times 10^{24}$	$\approx 9.7 \times 10^{30}$
n	10^6	3.6×10^9	$\approx 2.6 \times 10^{12}$	$\approx 3.12 \times 10^{15}$
$n \log n$	$\approx 10^5$	$\approx 10^9$	$\approx 10^1 1$	$\approx 10^{14}$
n^2	1000	6×10^4	$\approx 1.6 \times 10^6$	$\approx 5.6 \times 10^7$
n^3	100	≈ 1500	≈ 14000	≈ 1500000
2^n	19	31	41	51
$n!$	9	12	15	17

Example 2.38. ***Show that $n^3 \log n$ is $\Omega(n^3)$.***

Solution. By the definition of big-Omega, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $n^3 \log n \geq cn^3$ for $n \geq n_0$. Choosing $c = 1$ and $n_0 = 2$, shows $n^3 \log n \geq cn^3$ for $n \geq n_0$, since $\log n \geq 1$ in this range.

Example 2.39. An array A contains $n - 1$ unique integers in the range $[O, n - 1]$, that is, there is one number from this range that is not in A. Design an $O(n)$-time algorithm for finding that number. You are allowed to use only $O(1)$ additional space besides the array A itself.

$$\max(f(n), g(n)) = \theta(f(n) + g(n))$$

7. Let $f(n)$ and $g(n)$ be asymptotically non negative functions. If $f(n) = \theta(g(n))$, then $g(n) = \theta(f(n))$ prove or disprove.
8. Prove or disprove :
   (i) $O(2^{\log_a n}) = O(2^{\log_a n})$
   (ii) If $a < b$ then $n^a/2^n \in o(n^b)$
9. Suppose we have n total elements divided into m sorted lists. Explain how to produce a single sorted list of all n elements in time $O(n \lg n)$.
10. A computer performs 10^9 operations per second, what is the largest problem you can solve in an hour if our algorithm takes 3^n operations on problems of size n ?
11. Prove $\sum_{i=1}^{n} \frac{i}{2} = O(n)$
12. Show that 2^{n+1} is $O(2^n)$.
13. Show that $n^3 \log n$ is $\Omega(n^3)$.
14. Show that $\lceil f(n) \rceil$ is $O(f(n))$ is $f(n)$ is a positive nondecreasing function that is always greater than 1.
15. Define omega notation. Explain the terms involved in it. Give an example.
16. Write the non-recursive algorithm for finding the Fibonacci sequence and derive its time complexities.
17. Consider a polynomial in n of the form

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_m n^m + a_{m-1} n^{m-1} + a_2 n^2 + a_1 n + a_0 \text{ where } a_m > 0,$$

then $f(n) = \Omega(n^m)$.

18. Show that $f_1(n) \times f_2(n) = O((g_1(n) \times g_2(n))$ where $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$.
19. Prove that $f(n) = O(h(n))$ where $f(n) = O(g(n))$ and $g(n) = O(h(n))$.

QUESTIONS WITH ANSWERS

Q. 1. For the given x and n, write an algorithm to compute $x^n/n!$, and compute the running time.

Ans.

$$f(n) = \frac{x^n}{n!} = \frac{x}{1} \times \frac{x}{2} \times \frac{x}{3} \times \ldots \times \frac{x}{n}$$

Algorithm (x, n)

1. $fn = x$
2. For $k = 2$ to n do
3. $\quad fn = fn * x/k$
4. Endfor
5. Return fn

Line-1 and line-5 count for one unit each. Line-3 counts for three units per times executed (one multiplication, one division, and one assignment) and is executed $n - 1$ times, for a total of $3(n - 1)$ units. Line-2 has the hidden costs of initializing k, testing $k \le n$, and incrementing k. The total of all these is 1 to initialize, n for all the tests, and $n - 1$ for all the increments, which is $2n$. Thus, the run time of this algorithm is given by : $T(n) = 2 + 3(n - 1) + 2n = 5n - 1$.

Q. 2. Define omega notation. Explain the terms involved in it. Give an example.

Ans. Omega-notation provides an asymptotic lower bound. Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants $c > 0$ and $n_0 \ge 1$ such that

$$f(n) \ge cg(n) \text{ for all } n, \quad n \ge n_0$$

For example, let

$$f(n) = 2n^2 + 7n - 10 \text{ and } g(n) = n^2.$$

Since with $c = 1$ and for $n \ge n_0 = 2$, we have $2n^2 + 7n - 10 \ge cn^2$.

Verification : $2(2^2) + 7(2) - 10 \ge 1(2^2)$.

$$8 + 14 - 10 = 12 \ge 4.$$

Hence, $f(n) \ge cg(n)$.

Q. 3. Write the non-recursive algorithm for finding the Fibonacci sequence and derive its time complexity.

Ans. Algorithm (n)

Algorithm (n)	Count of primitive operations
1. $f_1 = 0;\ f_2 = 1$	2
2. print f_1, f_2	1
3. for $i = 3$ to n do	$1 + (n - 2) + (n - 1)$
4. $\quad f_3 = f_1 + f_2$	$2(n - 2)$
5. $\quad$ print f_3	$(n - 2)$
6. $\quad f_1 = f_2$	$(n - 2)$
7. $\quad f_2 = f_3$	$(n - 2)$
8. Endfor	
9. Return	

Total number of primitive operations = $T(n) = 7n - 9 \rightarrow O(n)$.

Q. 4. Write an algorithm to compute x^n for any real number x and integer n (where n is an integral power of 2). Prove that the running time of this algorithm is $\Theta(\log_2 n)$.

Ans. Algorithm that computes x^n for any real number x and integer n (where n is an integral power of 2).

Algorithm (x, n)

```
1. k = log2 (n)      // If n = 8 (or 2^3), then k = log2 (8) = 3
2. y = x
3. for i = 1 to k do
4.      y = y * y
5. Endfor
6. Return y
```

If $n = 2^k$ for some integer k, then $k = \log_2(n)$. Hence, the running time of this algorithm is $\Theta(\log_2 n)$.

Q. 5. Derive the function $f(n) = 12n^2 + 6n$ is $o(n^3)$ and $\omega(n)$.

Ans. The function $f(n) = o(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = 12n^2 + 6n \text{ and } g(n) = n^3$$

$$\lim_{n\to\infty} \frac{12n^2 + 6n}{n^3} = 0, \text{ and also } f(n) < cg(n).$$

"$f(n) = o(g(n))$" means for all $c > 0$ there exists some $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0$.

For $n \ge 7$ and $c = 2$, $f(n) = 12n^2 + 6n = 630$, and $cg(n) = 2n^3 = 686$.

$$\text{Hence, } f(n) = 12n^2 + 6n \text{ is } o(n^3).$$

The function $f(n) = \omega(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

$$f(n) = 12n^3 + 6n \text{ and } g(n) = n$$

$$\lim_{n\to\infty} \frac{n}{12n^2 + 6n} = 0 \text{ and also } f(n) > cg(n).$$

For $n \ge 2$ and $c = 18$, $f(n) = 12(2^2) + 6(2) = 60$, and $cg(n) = 18(2) = 36$.

Hence, $f(n) = 12n^2 + 6n$ is $\omega(n)$.

Q. 6. Show that $n^3 \log n$ is $\omega(n^3)$.

Ans. Let $f(n) = n^3 \log_2 n$ and $g(n) = n^3$

$f(n)$ is $\omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) > cg(n)$, for all $n \ge n_0$. Alternatively, $f(n)$ is $\omega(g(n))$, if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$$\lim_{n\to\infty} \frac{n^3 \log_2 n}{n^3} = \lim_{n\to\infty} \log_2 n = \infty \text{ for sufficiently large values of } n.$$

Hence, $n^3 \log_2 n$ is $\omega(n^3)$.

Q. 7. Consider a polynomial in n of the form

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_m n^m + a_{m-1} n^{m-1} + a_2 n^2 + a_1 n + a_0 \text{ where } a_m > 0,$$

then $f(n) = \Omega(n^m)$.

Ans. Consider a polynomial in n of the form

$$f(n) = \sum_{i=0}^{m} a_i n^i = a_m n^m + a_{m-1} n^{m-1} + a_2 n^2 + a_1 n + a_0, \text{ then } f(n) = O(n^m).$$

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

$$\le \sum_{i=0}^{m} |a_i| n^{i-m+m}$$

m is added and subtracted to the power of n.

$$\le \sum_{i=0}^{m} |a_i| n^{i-m} n^m$$

$$\le n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\le n^m \left[|a_0| n^{-m} + |a_1| n^{1-m} + |a_2| n^{2-m} + \ldots + |a_m| n^{m-m} \right]$$

$$\le n^m \left[|a_0| + |a_1| + |a_2| + \ldots + |a_m| \times 1 \right]$$

$n^{-\text{ve value}}$ are neglected for $n \ge 1$.

$$\le n^m \sum_{i=0}^{m} |a_i|$$

Hence, $f(n) = O(n^m)$ assuming m is fixed.

Q. 8. Show that $f(n) + g(n) = O(n^2)$ where $f(n) = 3n^2 - n + 4$ and $g(n) = n \log n + 5$.

Ans. Show that $f(n) = 4n^2 - 6n + 288 = \Omega(n^2)$.

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there are positive constants $c > 0$ and $n_0 \ge 1$ such that

$$f(n) = 4n^2 - 64n + 288, \text{ and } g(n) = n^2$$

This is true for $c = 1$ and $n_0 \ge 15 \rightarrow 4(15)^2 - 64(15) + 288 \ge 1(15^2)$.

This is $228 \ge 225$. Hence, $f(n) = 4n^2 - 64n + 288 = \Omega(n^2)$.

Q. 9. Show that $f_1(n) \times f_2(n) = O((g_1(n) \times g_2(n))$ where $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$.

Ans. For all $n \ge 1$, $f_1(n) \le \max(g_1(n), g_2(n))$ and $f_2(n) \le \max(g_1(n), g_2(n)$.

Therefore, $\quad f_1(n) + f_2(n) \le \max(g_1(n), g_2(n)) + \max(g_1(n), g_2(n))$

$f_1(n) + f_2(n) \le 2 \max(g_1(n), g_2(n))$ [By definition, $f(n) \le cg(n)$]

Thus, $\quad f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))$.

Q. 10. Let $f(n)$ and $g(n)$ be asymptotically non negative functions, using the basic definition of Θ notation prove that

$$\max(f(x), g(n)) = \Theta(f(n) + g(n))$$

Ans. $\max(f(x), g(n)) = \Theta(f(n) + g(n))$ is true if there exist C_1 . C_2 and $n_0 > 0$ such that $0 \le C_1 (f(n) + g(n))$ $\le \max(f(n) + g(n)) \le C_2 (f(n) + g(n))$ for all $n \ge n_0$.

Let us take $C_1 = \frac{1}{2}$, two situations may arise.

1. $f(n) > g(n) \Rightarrow \frac{1}{2}(f(n) + g(n)) < \frac{1}{2}(f(n) + f(n)) = f(n) = \max(f(n), g(n))$

2. $f(n) < g(n) \Rightarrow \frac{1}{2}(f(n) + g(n)) < \frac{1}{2}(f(n) + g(n)) = g(n) = \max(f(n), g(n))$

Now let us take $C_2 = 1$

$\max(f(n) + g(n)) \le 1\,(f(n) + g(n)) = f(n) + g(n)$

$\max(f(n) + g(n) \le f(n) + g(n)$

So by taking $C_1 = \frac{1}{2}$ and $C_2 = 1$ the given statement holds.

Q. 11. Prove $3n + 2 = O(n^2)$.

Ans.

$$\lim_{n\to\infty} \frac{3n+2}{n^2} = \lim_{n\to\infty}\left(\frac{3}{n} + \frac{1}{n^2}\right) = 0$$

$$3n + 2 = O(n^2) \text{ holds.}$$

OBJECTIVE TYPE QUESTIONS

1. The most popular asymptotic notations is
(a) Big-oh (b) Theta
(c) Little-oh (d) Omega

2. $\sum_{1\le k\le n} O(n)$, where $O(n)$ stands for order n is
(a) $O(n)$ (b) $O(n^2)$
(c) $O(n^3)$ (d) $O(3n^2)$

3. What is the running time to retrieve an element from an array of size n?
(a) $O(n-1)$ (b) $O(n)$
(c) $O\left(\frac{n}{2}\right)$ (d) $O(1)$

4. Retrieval from a linked list of size n has running time :
(a) $O(n-1)$ (b) $O(n)$
(c) $O\left(\frac{n}{2}\right)$ (d) $O(1)$

5. Which one of the following shows the correct relationship among some of the more comm[on] time complexities for the algorithms ?

(a) $O(\log_2 n) < O(n) < O(n \log_2 n) < O(2^n) < O(n^2)$

(b) $O(n) < O(\log_2 n) < O(n \log_2 n) < O(2^n) < O(n^2)$

(c) $O(n) < O(\log_2 n) < O(n \log_2 n) < O(2^n) < O(n^2)$

(d) $O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(2^n)$

6. Let $f(n) = n^2 \lg n$ and $g(n) = n(\lg n)^{10}$ be two positive functions of n. Which one of the followi[ng] statements is correct ?

(a) $f(n) = O(g(n))$ and $g(n) \neq Of(n))$

(b) $g(n) = O(f(n))$ and $f(n) \neq O(g(n))$

(c) $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$

(d) $f(n) = O(g(n))$ and $g(n) = O(f(n))$

7. The running time of the following algorithm

```
Procedure A(n)
If n ≤ 2 return 1;
Else return A(ceil (√n));
```

is best described by

(a) $O(n)$ (b) $O(\log_2 n)$

(c) $O(\log \log_2 n)$ (d) $O(1)$

8. Consider the following three claims :

(I) $(n + k)^m = \Theta(nm)$ where k and m are constants

(II) $2^{n+1} = O(2^n)$

(III) $2^{2n+1} = O(2^n)$

Which of these claims are correct ?

(a) (I) and (II) (b) (I) and (II)

(c) (II) and (III) (d) (I), (II) and (III)

9. The tightest lower bound on the number of comparisons, in the worst-case, for comparison-base[d] sorting is of the order of

(a) n (b) n^2

(c) $n \log n$ (d) $n \log_2 n$

10. What does the following algorithm approximate ? Assume $m > 1$ and $\varepsilon > 0$.

```
x = m;
y = 1;
while (x - y > ε)
{  x = (x + y)/2;
   y = m/x;
}
print(x);
```

(*a*) $\log_2 m$

(*b*) m^2

(*c*) $m^{1/2}$

(*d*) $m^{1/3}$

11. Consider the following program segment :

```
int j, n;
j = 1;
while (j <= n)
    j = j*2;
```

The number of comparisons made in the execution of the loop for any $n > 0$ is :

(*a*) $\lfloor \log_2 n \rfloor + 1$

(*b*) n

(*c*) $\lfloor \log_2 n \rfloor$

(*d*) $\lfloor \log_2 n \rfloor + 1$

12. The minimum number of comparisons required to determine if an integer appears more than $n/2$ times in a sorted array of n integers is

(*a*) $\Theta(n)$

(*b*) $\Theta(\log n)$

(*c*) $\Theta(1)$

(*d*) None of these

ANSWERS

1. (*a*)	**2.** (*b*)	**3.** (*d*)	**4.** (*b*)	**5.** (*d*)
6. (*a*)	**7.** (*a*)	**8.** (*a*)	9. (a)	**10.** (*a*)
11. (*a*)	**12.** (*a*)			

Chapter 3

Recurrences

3.1. RECURRENCE RELATIONS

INSIDE THIS CHAPTER

An algorithm is recursive if it can call itself. Recursive algorithms generally solve a problem by reducing the problem to an instance of the same problem with smaller input. Often times, a recursive solution to a problem is the most natural, and therefore gives the simplest procedure. We note the following two points.

(*a*) Always identify the base case and associated result first.

(*b*) Make sure the recursive call is for a smaller problem.

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A recurrence relation is an equation or inequality that describes a function in terms of its value on smaller inputs. Let us consider the classical example of computing the factorial of a number.

Example 3.1. The factorial function is given as, $f(n)$:

$$n! = \quad n \times (n-1) \times (n-2) \times \ldots. \times 2 \times 1, \text{ where } n \text{ is an integer}$$

Let $T(n)$ be the running time for the following algorithm

Algorithm 3.1. Factorial (n) ....

```
1. If n = 0, then
2.     Return 1
3. else
4.     Return n * factorial (n - 1)
5. End if
```

Running time for lines 1 and 2 is $O(1)$, and for line 4 is $O(1) + T(n-1)$. The factorial(n) is called $(n-1)$ times.

Thus, for some constants 'c' and 'd',

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ c + T(n-1), & \text{if } n > 1 \end{cases}$$

We substitute in the above recurrence, $n = 1, 2, ..., (n - 1)$.

$$\begin{aligned} T(1) &= d \\ T(2) &= c + T(1) = c + d \\ T(3) &= c + T(2) = 2c + d \\ &\vdots \\ T(n-1) &= c + T((n-1)-1) = c + T(n-2) \\ T(n) &= c + T(n-1) = c + [c + T(n-2)] = 2c + T(n-2) \end{aligned}$$

and so on. In general,

$$T(n) = ic + T(n-i) \quad \text{if } n > i$$

Finally, when $i = n - 1$, we get

$$\begin{aligned} T(n) &= ic + T(n-i) = (n-1)c + T(n-(n-1)) \\ &= c(n-1) + T(1) \\ T(n) &= c(n-1) + d \end{aligned}$$

from the above, we can conclude $T(n)$ is $O(n)$.

3.1.1. Techniques Used in Recurrence Relation

To solve any given recurrence relation. We can use either of the following four techniques:

(*a*) Substitution Method

(*b*) Iterative Method

(*c*) Recurrence Tree Method

(*d*) Master Method

3.2. SUBSTITUTION METHOD

It is also known as bounding recurrence relation. When analyzing a recurrence relation that depicts the resources used by an algorithm, we are interested in finding the tightest possible asymptotic bound on closed form. Thus, instead of determing the exact closed form, and then expressing this solution using asymptotic notation, it is often easier to make a guess directed at the asymptotic form, and then verify this guess using induction called constructive induction, the steps followed are :

(*a*) Guess a particular solution in asymptotic form for the given recurrence relation.

(*b*) Verify the guessed solution using mathematical induction and find the value of constant C.

Example 3.2. Consider the recurrence relation.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n, & \text{if } n > 1 \end{cases} \quad ...(1$$

Solution. If we wish to prove that $T(n) = O(n^2)$, then it is sufficient to show that $T(n) \leq Cn^2$ f some positive constant $C > 0$ and for all values of $n > n_0$ (n_0 is a positive constant). First, we w assume

$$T(\alpha) \leq C\alpha^2 \text{ for } 1 \leq \alpha \leq n-1.$$

Then we will use this induction hypothesis to show that $T(n) \leq Cn^2$. By substituting the induction hypothesis into equation (1) we get,

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\leq C(n-1)^2 + n \qquad \text{if } T(n) \leq Cn^2 \text{ then } T(n-1) \leq C(n-1)^2 \\ &= C(n^2 - 2n + 1) + n \\ &= Cn^2 - 2Cn + C + n \\ \therefore \quad T(n) &\leq Cn^2 \end{aligned}$$

$T(n) \leq Cn^2$ holds for all $n \geq 1$ when $C = 1$. This means that $n_0 \geq 1$. From equation (1), we have that $T(1) = 1$, and from the induction hypothesis $T(1) = C(1)^2$. Choosing $C = 1$ will also satisfy this base case, and hence $T(n) = O(n^2)$.

3.2.1. How to Make a Good Guess

For the given recurrence relation *i.e.,*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case I : If $a = b = 2$, then

$$T(n) = O(f(n) \log n) \text{ is a good guess.}$$

Case II : If $a = 1$, $b =$ any value, then

$$T(n) = O(\log n) \text{ is a good guess.}$$

Case III : if $a \neq 1$, $b > a$, then

$$T(n) = O[f(n).\ n] \text{ is a good guess.}$$

3.2.2. Bad Guess

1. Let us gness $O(n)$ for the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + 1$$

thus we need to prove $T(n) \leq Cn$,

Substituting this we get

$$\begin{aligned} T(n) &\leq 2C[[\lfloor n/2 \rfloor] + 1 \\ &\leq \frac{Cn}{2} + 1 \\ &= Cn + 1 \end{aligned}$$

for any value of C, $T(n)$ is not less than Cn, because of 1. Hence it is a bad guess.

2. We make a larger guess $T(n) = O(n^2)$, thus.

$$\begin{aligned} T(n) &\leq 2C\frac{n^2}{2} + 1 \\ &= Cn^2 + 1 \\ &= Cn^2 \end{aligned}$$

if $C \geq 1$, for large value of n this relation will hold as n grows much father than 1. Some can ignore this value. $T(n) = O(n^2)$ will hold.

3. Now we make an intermediate guess such that $O(n)$ should hold. Let $T(n) \leq Cn - b$, where $b \geq 0$ then

$$T(n) \leq 2\left(\frac{Cn}{2} - b\right) + 1$$

$$= Cn - 2b + 1 \leq Cn - b,$$

for $b \geq 1$ (if $b = 1$, b cancels out with 1). Since b is constant this of order $O(n)$.

4. Let us guess $O(n)$ in the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

Thus we need to prove $T(n) \leq Cn$, substituting

$$T(n) \leq 2C\left(\frac{n}{2}\right) + n = Cn + n$$

$T(n) \neq Cn$ as addition term is a variable, hence $T(n) \neq O(n)$ in this recurrence.

5. We have $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$

here we can choose another variable, say

$$m = \lg n \Rightarrow n = 2^m,\ n^{1/2} = 2^{m/2}$$

Thus, $T(2^m) = 2T\,(2^{m/2}) + m$

We rename $T(2^m) = S(m)$,

$$S\left(\frac{m}{2}\right) = T(2^{m/2})$$

Hence new recurrence is

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

As 2^m is written as m in new notation S, hence we can write $T(2^{m/2})$ as $S\left(\frac{m}{2}\right)$ but above recurrence no solution.

$O(m \lg m)$. Substituting form we get $O(\lg n \lg \lg n)$.

3.2.3. Examples

Example 3.3. Solve the recurrence

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \text{ using substitution method.}$$

Solution. Let us guess a solution

$$T(n) = O(n \log n)$$

i.e., $T(n) \leq Cn \log n \;\; \forall \;\; n \geq n_0$

Substitute it in the equation *i.e.*, we get

$$T(n) \leq 2C\left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + n$$

$$\leq Cn \log \frac{n}{2} + n$$

NOTE

As n are the number of elements, which is very large, therefore $\frac{n}{2}$ is also very large.

Hence $\frac{n}{2} \cong \left\lfloor \frac{n}{2} \right\rfloor \cong \left\lceil \frac{n}{2} \right\rceil$

Now,
$$T(n) = Cn \log n - Cn \log 2 + n$$
$$= Cn \log n - C + n \qquad \text{as } \log_2 2 = 1$$
$$T(n) \leq Cn \log n, \quad C \leq 1$$

By induction hypothesis we require to show that our solution holds for the boundary condition, *i.e.*,

$$T(1) \leq C(1)\ (\log 1) = 0 \qquad \text{as } \log_2 1 = 0$$

Thus for any value of C, this will not hold, so by asymptotic definition we need to prove.

$$T(n) \leq Cn \log n \quad \forall\ n \geq n_0$$
$$T(2) \leq C(2)\ (\log 2),$$
$$T(3) \leq C(3)\ (\log 3)$$

as 'n' can't be 1, this relation holds for $C \geq 2$

Thus, $T(n) = O(n \log n)$ is true.

Example 3.4. Show that solution $T(n) = (\lfloor n/2 \rfloor) + 1$ is O (lg n).

Solution. Let us guess that solution is $O(\lg n)$, thus we need to prove $T(n) \leq C \lg n$.

Substituting this we get,

$$T(n) \leq C \log \frac{n}{2} + 1$$
$$= C \lg n - C \lg 2 + 1$$
$$= C \lg n - C + 1$$
$$\leq C \lg n \text{ for } C \geq 1$$

Hence, $T(n) = O(\lg n)$ for $C \geq 1$

Example 3.5. Solve $T(n) = 2T(\sqrt{n}) + 1$.

Solution. Let $m = \lg n$, s.t. $n = 2^m$, $n^{1/2} = 2^{m/2}$

Thus, $T(2^m) = 2T(2^{m/2}) + 1$

Changing the recurrence to $S(m) = T(2^m)$.

We get, $S(m) = 2S\left(\frac{m}{2}\right) + 1$

Solution to this $O(\lg m)$.

Therefore $T(n) = 0$ (lg lg n).

Example 3.6. Show that solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

Solution. Let us guess solution is $O(n \lg n)$ then we have to prove $T(n) \leq Cn \lg n$, substituting, we get,

$$
\begin{aligned}
T(n) &\le 2\left(\frac{Cn}{2}\lg\left(\frac{n}{2}\right)+17\right)+n \\
&= Cn\lg\left(\frac{n}{2}\right)+34+n \\
&= Cn \lg n - Cn \lg 2 + 34 + n \\
&= Cn \lg n - Cn + 34 + n \\
&= Cn \lg n - (C-1)n + 34 \\
&= Cn \lg n - bn + 34 \le Cn \lg n
\end{aligned}
$$

if $C \ge 1$, where b is constant.

Hence, $\quad T(n) = O(n \lg n)$

Example 3.7. Show that solution of $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $\Omega(n \lg n)$. Conclude that the solution is θ $(n \lg n)$.

Solution. Let the solution to $T(n)$ is $\Omega(n \lg n)$ then we need to prove $T(n) \ge Cn \lg n$.

$$
\begin{aligned}
T(n) &\ge 2\left(C\frac{n}{2}\lg\frac{n}{2}\right)+n \\
&\ge Cn\lg\frac{n}{2}+n \\
&= Cn \lg n - Cn \lg 2 + n \\
&= Cn \lg n - Cn + n \\
&\ge Cn \lg n \text{ for } C \ge 1
\end{aligned}
$$

Therefore $T(n) = \Omega(n \lg n)$, for $C \le 1$ but for large values of n, and $CT(n) \le Cn \lg n$ also holds hence $T(n) = \Theta(n\ lg\ n)$.

Example 3.8. Show that solution to $T(n)=2T\left(\left\lfloor \frac{n}{2} \right\rfloor+27\right)+n$ is $O(n \log n)$.

Solution. Let us guess solution is $O(n \lg n)$, then we have to prove $T(n) \le Cn \lg n$, substituting we get,

$$T(n) \le \left[2\left(\left\lfloor \frac{n}{2} \right\rfloor+27\right)\lg\left(\left\lfloor \frac{n}{2} \right\rfloor+27\right)+n\right]$$

As n is the number of element and which is very large hence,

$$\frac{n}{2} \cong \left\lfloor \frac{n}{2} \right\rfloor \cong \left\lceil \frac{n}{2} \right\rceil$$

and adding a constant value to this large value $\frac{n}{2}$ does not effect its actual value, hence

$$\left\lfloor \frac{n}{2} \right\rfloor \cong \frac{n}{2}$$

we get $\quad T(n) \le 2C\frac{n}{2}\lg\frac{n}{2}+n$

$$= Cn \log n - Cn \lg 2 + n$$
$$= Cn \lg n - Cn + n$$
$$T(n) \le Cn \lg n, \ \forall \ C \le 1$$

Hence $T(n) = O(n \lg n)$ is true.

3.3. ITERATIVE METHOD

This technique is also known as backward substitution. In this method, we simply continue to substitute back until we see a pattern of some sort. We then deduce a formula from the pattern. Once we discover a likely formula we prove that the formula actually solves the recurrence relation.

Example 3.8. Solve the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases} \qquad \text{...(1)}$$

Solution. Since $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$, we have

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T\left(\frac{n}{4}\right) + 2n$$

Substituting for $T\left(\frac{n}{4}\right)$, we obtain

$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

Continuing in this manner, we find that

$$T(n) = 2^K T\left(\frac{n}{2^K}\right) + Kn$$

After $\log_2 n$ substitution ($K = \log_2 n$), we have

$$T(n) = 2^{\log_2 n} \times T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \times n.$$

Noting that $2^{\log_2 n} = n$, we have

$$T(n) = n \times T\left(\frac{n}{n}\right) + n \times \log_2 n$$
$$T(n) = nT(1) + n \log_2 n$$
$$T(n) = n + n \log_2 n \text{ as } T(1) = 1$$

Example 3.9. Solve the following recurrence using iteration.

$$T(n) = 3T\left(\frac{n}{4}\right) + n \qquad \text{...(1)}$$

Solution. Since $T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{16}\right)+\frac{n}{4}$

Substitute $T\left(\frac{n}{4}\right)$ in equation (1)

$$T(n) = 3\left[3T\left(\frac{n}{16}\right)+\frac{n}{4}\right]+n = 9T\left(\frac{n}{16}\right)+\frac{3n}{4}+n$$

Continuing in this manner we get

$$T(n) = n+\frac{3n}{4}+\frac{9n}{16}+\ldots+3^i\,T\left(\frac{n}{4^i}\right).$$

Assume $\frac{n}{4^i} = 1$ since $T(1) = 1$

i.e. $\quad i = \log_4 n$

Substitute $i = \log_4 n$, we have

$$T(n) = n+3\frac{n}{4}+9\frac{n}{16}+\ldots+3^{\log_4 n}\,T(1)$$

$$= \sum_{i=0}^{\log_4 n}\frac{3^i}{4^i}\times n+\Theta(n^{\log_4 3}) \text{ as } a^{\log_b n} = n^{\log_b a}$$

$$\leq \sum_{i=0}^{\infty} n\times\left(\frac{3}{4}\right)^i+\Theta(n^{\log_4 3})$$

$$T(n) \leq n\times\frac{\left(\frac{3}{4}\right)^0}{1-\frac{3}{4}}+\Theta(n^{\log_4 3})$$

$$\leq 4n+\Theta(n^{\log_4 3})$$

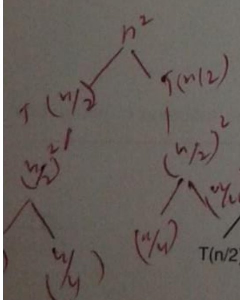
$$\leq 4n+\Theta(n) \text{ as } \log_4 3<1$$

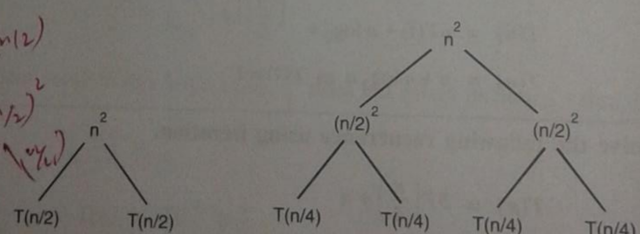
$$T(n) = O(n) \text{ as } [O(n)+\Theta(n) = O(n)]$$

3.4. RECURRENCE TREE METHOD

We can visualise iteration method as recursion tree in which at each level nodes are expanded. We consider second term in recurrence as root. It is useful when divide and conquer algorithm is used.

$$T(n) = 2T\left(\frac{n}{2}\right)+n^2$$





(a) (b)

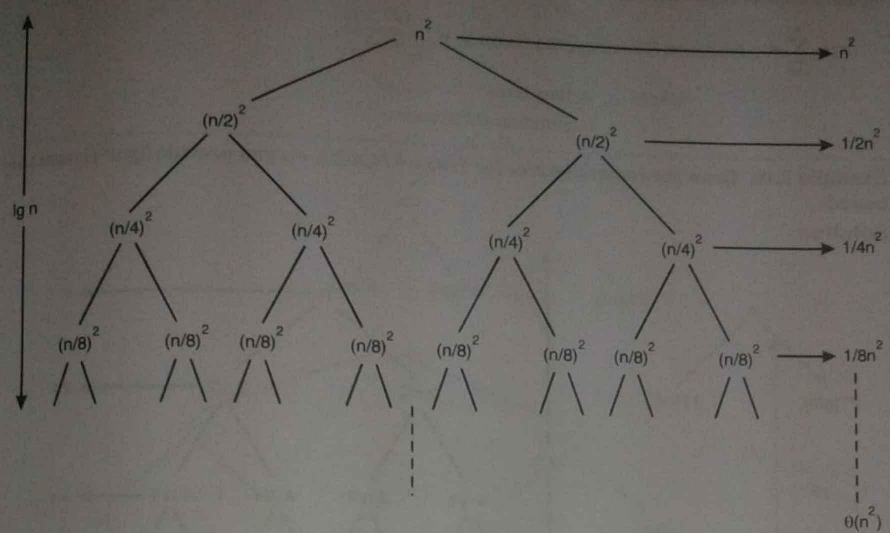


Fig. 3.1

Hence solution is $\theta(n^2)$.

Summing the values at each level, we get

$$\begin{aligned} n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots. &= n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots.\right) \\ &= n^2 \sum_{i=0}^{\lg n} \frac{1}{2^i} \leq n^2 \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= n^2 \frac{\frac{1}{2^0}}{1 - \frac{1}{2}} \\ &= 2n^2 \end{aligned}$$

Hence, solution is $\theta(n^2)$.

Useful Formulae :

(1) $\sum_{i=1}^{n} \frac{1}{a^i}$ if common ratio is less than 1, then convert it in infinite GP series *i.e.* $\sum_{i=1}^{\infty} \frac{1}{a^i}$.

(2) $\sum_{i=0}^{\infty} \frac{1}{a^i}$, for infinite GP series, the formula is $\frac{a}{1-r}$

where a = first term

r = common difference

(3) $\sum_{i=0}^{n} a^i$, for finite GP series, the formula is $\frac{a(r^n - 1)}{r - 1}$,

where a = first term

r = common difference

Example 3.10. Draw the recurrsion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + n$ and provide light asymptotic bounds.

Solution.

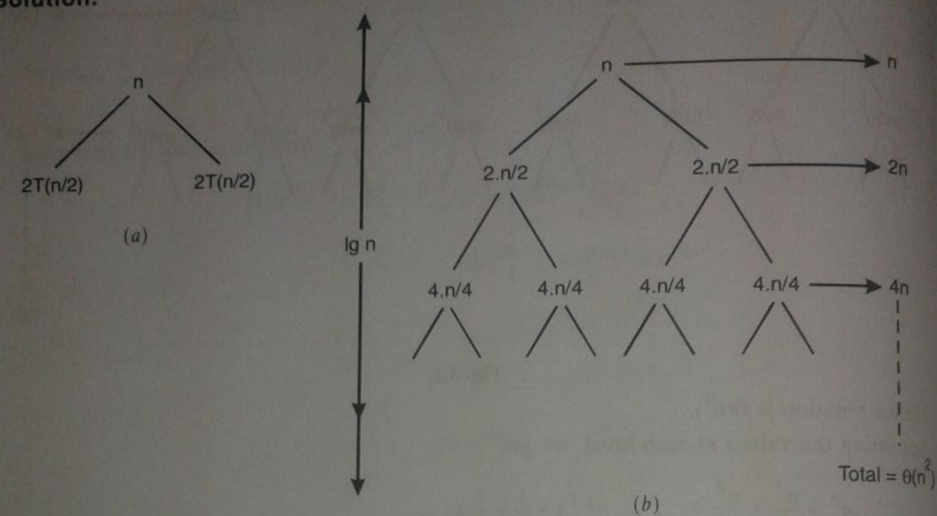


Fig. 3.3

$n + 2n + 4n + \ldots$ lg n times

$$n(1 + 2 + 4) + \ldots \text{ lg } n \text{ times}) = n\sum_{i=0}^{\lg n} 2^i = n \cdot \frac{(2^{\log_2 n} - 1)}{2 - 1} = n^2 - n = \Theta(n^2)$$

NOTE

Height of tree with n nodes is log n.

Example 3.11. Solve the recurrence $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$.

Solution.

$$T(n) = n + \left(\frac{n}{3} + \frac{2n}{3}\right) + \left(\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9}\right) + \ldots$$

$$= n + n + n + \ldots (\log_{3/2} n + 1) \text{ times}$$

$$= \sum_{i=0}^{\log_{3/2} n} n = \Theta(n \log_{3/2} n)$$

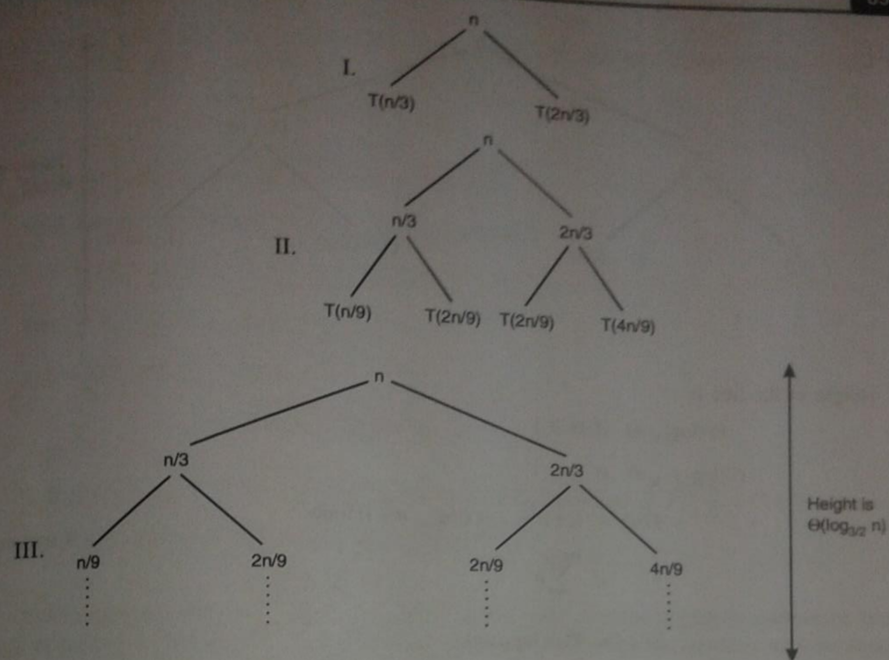
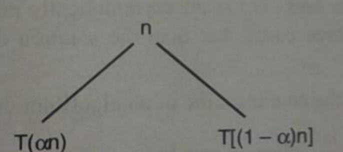


Fig. 3.4.

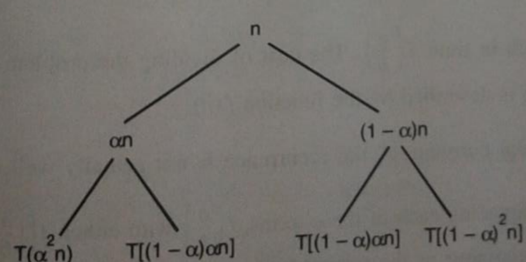
Example 3.12. Solve the following recurrence

$$T(n) = T(\alpha n) + T[(1-\alpha)n] + n, \ |\alpha| \geq 1.$$

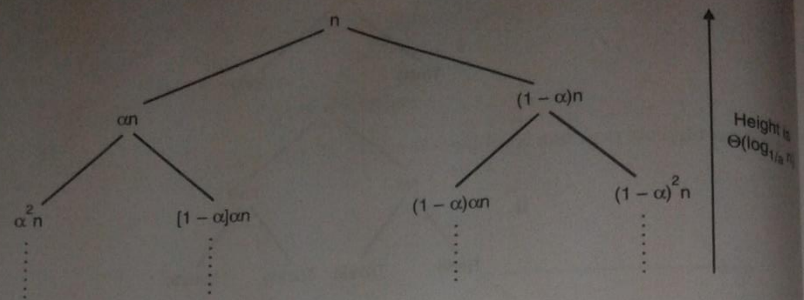
Solution. I.



II.



III.



Height of the tree is

$$\Theta(\log_{1/\alpha} n) \quad \text{if } \alpha \geq 1$$

$$\Theta(\log_{1/1-\alpha} n) \quad \text{if } \alpha < 1$$

$$T(n) = n + n + \ldots. + (\log_{1/\alpha} n + 1) \text{ time}$$

$$= \sum_{i=0}^{\log_{1/\alpha} n} n$$

$$= \Theta(n \log_{1/\alpha} n).$$

3.5. MASTER METHOD

The master method provides a "Cook book" method for solving recurrences of the form :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. The master method requires memorization of three cases, but then the solution of many recurrences can be determined quite easily.

The above recurrence describes the running time of an algorithm that divides a problem of size n into 'a' subproblems each of size $\frac{n}{b}$, where a and b are positive constants. The 'a' subproblems are solved recursively, each in time $T\left(\frac{n}{b}\right)$. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

As a matter of technical correctness, the recurrence is not actually well defined because $\frac{n}{b}$ might not be an integer. Replacing each of the a terms $T\left(\frac{n}{b}\right)$ with either $T\left(\left\lfloor \frac{n}{b} \right\rfloor\right)$ or $T\left(\left\lceil \frac{n}{b} \right\rceil\right)$ does not effect the aymptotic behaviour of the recurrence.

3.5.1. The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence,

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

There we interpret $\frac{n}{b}$ to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$.

Then $T(n)$ can be bounded asymptotically as follows :

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,

then $$T(n) = \Theta(n^{\log_b a})$$

2. If $f(n) = \Theta(n^{\log_b a})$ then

$$T(n) = \Theta(n^{\log_b a} \log_2 n)$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$, for some constant $\epsilon > 0$. Then and if $af\left(\frac{n}{b}\right) \leq C \times f(n)$ for some constant $C < 1$, and all sufficiently large n, then

$$T(n) = \Theta(f(n)).$$

Before applying the master theorem, let us spend some moment trying to understant what it says. In each of three cases, we are comparing the function $f(n)$ with the function with the function $n^{\log_b a}$. If as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If as in case 3, the function $f(n)$ is the larger, then solution is $T(n) = \Theta(f(n))$.

Example 3.13. Solve the recurrence using master method. $T(n) = 9T\left(\frac{n}{3}\right) + n.$

b

Solution. Here, $$a = 9, \quad b = 3, \quad f(n) = n$$

$$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 (3)^2} = n^{2\log_3 3} = n^2$$

Writing $f(n)$ in appropriate form

$$f(n) = O(n^{\log_3 9 - 1}) = O(n)$$

hence we can apply case (i)

So the solution is $\theta(n^{\log_b a})$ by case (i), *i.e.*,

$$\theta(n^{\log_3 9}) = \theta(n^2)$$

Example 3.14. Solve the recurrence by master method.

$$T(n) = T\left(\frac{2n}{3}\right) + 1.$$

Solution. Here, $a = 1$, $b = 3/2$, $f(n) = 1$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

which is equal to $f(n)$, hence case (*iii*) applies.

Thus solution is

$$T(n) = \theta(n^{\log_b a} . \lg n) = \theta(1 \lg n)$$
$$= \theta(\lg n).$$

Example 3.15. Solve the recurrence by master method.

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

Solution. Here, $a = 3, \quad b = 4, \quad f(n) = n \lg n$

$$n^{\log_a b} = n^{\log_4 3} = 0(n^{0.793})$$

$$f(n) = \Omega(n^{(\log_4 3 + \varepsilon)}) \text{ where } \varepsilon \approx 0.2, \text{ case 3 applies.}$$

Here we can see that regularity condition of case 3 holds.

i.e. $$af\left(\frac{n}{b}\right) \le Cf(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n\lg n = Cf(n) \text{ for } C = \frac{3}{4}$$

Therefore solution to the recurrence is

$$T(n) = \theta(n \lg n) = \theta(f(n))$$

Example 3.16. $T(n) = 2T\left(\frac{n}{2}\right) + n\lg n$.

Solution. $a = 2, b = 2, f(n) = n \lg n$

We have, $n^{\log_b a} = n^{\log_2 2} = n$

Thus $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a}$, *i.e.* n but it is not polynomially greater as the ratio of $\frac{f(n)}{n^{\log_b a}} = \lg n$ is asymptiotically less than n^{ε} for any +ve value of ε. Hence, Master theorem can not be applied.

Example 3.17. Solve $T(n) = 2T\left(\frac{n}{2}\right) + n^3$ **by master method.**

Solution. We have $a = 2$, $b = 2$, $f(n) = n^3$

$$n^{\log_b a} = n^{\log_2 2} = n$$

Case (*iii*), $f(n) = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{1+2})$, here $\varepsilon = 2$

and also, $2f\left(\frac{n^3}{2^3}\right) < cf(n^3)$, is true for $C = \frac{1}{4}$ hence regularity condition holds, therefore

$$T(n) = \theta(n^3)$$

Example 3.18. Solve $T(n) = T\left(\frac{9n}{10}\right) + n$ by master method, if possible.

Solution. Here, $a = 1$, $b = \frac{10}{9}$, $f(n) = n$

$$n^{\log_b a} = n^{\log_{10/9} 1} = n^0 = 1$$

We can write
$$f(n) = n = \theta(n) = \theta(n^{\log_b a + \varepsilon})$$
$$= \theta(n^{0+1}), \text{ where } \varepsilon = 1$$

So case (*iii*) applies

Also, $af\left(\frac{n}{b}\right) \le Cn$, as

$$f\left(\frac{9n}{10}\right) = \frac{9n}{10} \le Cn, \text{ for } C = \frac{9}{10}$$

Hence regularity condition is also satisified, therefore,

$$T(n) = \theta(f(n)) = \theta(n)$$

Example 3.19. Solve, $T(n) = 16T\left(\frac{n}{4}\right) + n^3$

Solution. By master theorem,

$$a = 16, b = 4, f(n) = n^3$$
$$n^{\log_b a} = n^{\log_4 16} = n^2 = O(n^2)$$
$$f(n) = n^3 = (n^{\log_b a + \varepsilon}) = (n^{2+1}) = \Omega(n^3) \text{ and } \varepsilon = 1$$

Also,
$$16f\left(\frac{n^3}{4^3}\right) \le Cf(n^3)$$
$$16.\frac{n^3}{64} \le Cn^3, \text{ for } C = \frac{1}{4}, \text{ it is true so regulartity condition holds.}$$

Hence, by case (*iii*) $T(n) = \theta(f(n)) = \theta(n^3)$

Example 3.20. Solve $T(n) = 3T\left(\frac{n}{2}\right) + n \log n$.

Solution. By master method,

$$a = 3, b = 2, f(n) = n \log n$$

$$n^{\log_b a} = n^{\log_2 3} = 0\,(n^{1.58})$$

$$f(n) = \Omega(n^{\log_2 3-\varepsilon}),\ \text{where } \varepsilon = 0.58$$

We are not sure whether we can the apply the master method.

As we have $f(n) = n \lg n$, and $n^{\log_b a} = n^{1.58}$, and also it is in proper form *i.e.* $a = 3$, $b = 2$, $f(n) = n \lg n$,

But $f(n)$ is asymptotically smaller than,

$$n^{\log_b a} = n^{1.58},\ i.e.,\ O(n^{1.58}) > O(n \log n).$$

but is not polynomically smaller than $n^{\log_b a}$ *i.e.*, $n \log n > n^{1.58 - 0.58}$. The ratio $\frac{f(n)}{n^{\log_b a}}$ is asymptotically greater than n^{ε} for some +ve constant ε. So the recurrence falls in the gap of case (*i*) and (*ii*).

Hence, $\quad T(n) = \Theta(n^{1.58})$

Example 3.21. Solve the following recurrence using master method.

(*a*) $\quad T(n) = \frac{1}{2}T\left(\frac{n}{2}\right) + f(n)$

(*b*) $\quad T(n) = 2T(n) + n$

(*c*) $\quad T(n) = 2T\left(\frac{n}{4}\right) - n^2$

(*d*) $\quad T(n) = 4T\left(\frac{n}{64}\right) + \frac{n}{\log_2 n}$

Solution.

(*a*) $a = \frac{1}{2}$, $b = 2$, $f(n) = f(n)$

as $a < 1$ but according to master theorem

$a \geq 1$, hence master theorem is not applicable.

(*b*) $a = 2$, $b = 1$, $f(n) = n$

as $b = 1$, but according to master theorem $b > 1$, hence master theorem is not applicable.

(*c*) $a = 2$, $b = 4$, $f(n) = -n^2$

as $f(n)$ is a –ve function, but according to the master theorem $f(n)$ should be bounded as a +ve function, hence master theorem is not applicable.

(*d*) $a = 4$, $b = 64$, $f(n) = \frac{n}{\log_2 n}$

According to master theorem, the function $f(n)$ in the relation should be of the form $n^K \log^K n$ where $K \geq 0$ but here $f(n) = n \log^{-1} n$ hence master theorem is not applicable.

SOLVED EXAMPLES

Example 3.22. Solve the recurrence, $T(n) = 7T\left(\frac{n}{2}\right) + n^2$.

Solution. By Master theorem,

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 7} = \theta(n^{2.8})$$

$$f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{2.8L - 0.8L}) = O(n^2)$$

There by case 1, we have

$$T(n) = O(n^{\log_b a}) = O(n^{2.81})$$

Example 3.23. Solve the recurrence, $T(n) = 7T\left(\frac{n}{3}\right) + n^2$.

Solution. Applying Master theorem

$$a = 7, \quad b = 3, f(n) = n^2$$

$$n^{\log_b a} = n^{\log_3 7} = O(n^{1.77})$$

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ to match with } n^2$$

$$\Omega(n^{1.77+0.23}), \varepsilon = 0.23$$

So case 3 can be applied, check for regularity condition,

$$af\left(\frac{n}{b}\right) \leq Cf(n)$$

$$7f\left(\frac{n^2}{3^2}\right) \leq Cf(n^2), \text{ is true for } C = \frac{7}{9}$$

hence case 3 holds

Thus, $\quad T(n) = \Theta(f(n)) = \Theta(n^2).$

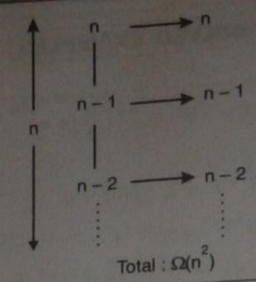
Example 3.24. Solve the recurrence by iteration and draw its recursion tress if possible.

Solution. $\quad T(n) = T(n-1) + n$, by iteration method.

$$= n + T(n-1)$$
$$= n + (n-1 + T(n-2))$$
$$= n + (n-1 + (n-2 + T(n-3)))$$
$$= n + n - 1 + n - 2 + n - 3 + ... + T(1)$$

We have $\quad T(1) = \theta(1)$

$$= (n + n + n + ... n \text{ times}) - (1 + 2 + 3 + ... n) + \theta(1)$$
$$= n^2 - n + \theta(1)$$
$$= \Omega(n^2)$$



Example 3.25. $T(n) = T(n-1) + \frac{1}{n}$.

Solution. By iteration method

$$\begin{aligned} T(n) &= \frac{1}{n} + T(n-1) \\ &= \frac{1}{n} + \left(\frac{1}{n-1} + T(n-2)\right) \\ &= \frac{1}{n} + \frac{1}{n-1} + \left(\frac{1}{n-2} + T(n-3)\right) \\ &= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \frac{1}{n-3} + \ldots T(1) \\ &= \sum_{i=0}^{n-2} \frac{1}{n-i} + T(1) \le \sum_{i=0}^{n} \frac{1}{n-i} + \theta(1) \end{aligned}$$

Let $n - i = x$, $-di = dx$, $di = -dx$

Thus, it can be transformed into an integeral

$$\sum_{i=0}^{n} \frac{1}{n-i} = -\int_{n}^{0} \frac{dx}{x} = Inx$$

Thus, $T(n) = \theta(In\, n) + \theta(1) = \theta(\lg n)$

Example 3.26. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$, **solve this recurrence.**

Solution.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n} \\ &\le 2T\left(\frac{n}{2}\right) - n \lg n = 2T\left(\frac{n}{2}\right) - \theta\,(n \lg n) \end{aligned}$$

This can't be solved by master method, solving the iteration method we have the summation

$$2T\left(\frac{n}{2}\right) = \left(\sum_{i=0}^{k-1} 2^i\, 2^{k-i} \log\,(2^{k-i})\right)$$

$$= 2^k \sum_{i=0}^{k-1} (k-i)$$

$$= 2^{k-1} k(k+1) = 0(n \log^2 n)$$

As $\quad k = \log n$

and this term is greater than $\theta(n \lg n)$ therefore solution to recurrence is $T(n) = O(n \log^2 n)$.

Example 3.27. Solve the recurrence $T(1) = 1$, $T(n) = 3T\left(\frac{n}{2}\right) + 2n^{1.5}$.

Solution. We have, $\quad T(n) = 3T\left(\frac{n}{2}\right) + 2n^{1.5} \quad \text{...}(i)$

For converting it to proper form divide both sides by 2.

$$\frac{T(n)}{2} = \frac{3}{2}T\left(\frac{n}{2}\right) + n^{1.5} \quad \text{...}(ii)$$

Let $\quad \frac{T(n)}{2} = U(n)$

$\Rightarrow \quad U\left(\frac{n}{2}\right) = \frac{1}{2}T\left(\frac{n}{2}\right)$

Substituting in (ii) we get,

$$U(n) = 3U\left(\frac{n}{2}\right) + n^{1.5} \quad \text{...}(iii)$$

and also, $\quad U(1) = \frac{1}{2}0.5$

Now, we can solve the recurrence equation (iii) by master method here,

$$a = 3, \quad b = 2, \quad f(n) = n^{1.5}$$

$$n^{\log_b a} = n^{\log_2 3} \approx n^{1.585} \approx n^{1.59}$$

By case 1, $\quad f(n) = \theta(n^{\log_b a - \varepsilon}) = \theta(n^{1.59-0.09}) \quad$ where, $\varepsilon = 0.09$

$$= \theta(n^{1.59})$$

Hence, $\quad U(n) = O(n^{1.59}) = O(n^{\log_2 3})$

Since, $\quad T(n) = 2.U(n)$

$$T(n) = O(n^{\log_2 3}).$$

Example 3.28. Solve the following recurrence, $T(n) = 4T\left(\frac{n}{2}\right) + n^3$.

Solution. By Master method,

$$a = 4, \quad b = 2, \quad f(n) = n^3$$

$$n^{\log_b a} = n^2 = O(n^2)$$

By case 3, $\quad f(n) = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{2+1}), \varepsilon = 1$

Thus, case 3 can be applied, now check for the regularity condition $4f\left(\frac{n^3}{2^3}\right) \le Cf(n^3)$, this is true for $C = \frac{1}{2}$ hence solution to this recurrence is $\theta(f(n))$

i.e., $$T(n) = \theta(n^3).$$

Example 3.29. Solve the recurrence, $T(n) = 4T\left(\frac{n}{2}\right) + n^2$.

Solution. By master method,

$$a = 4,\ b = 2,\ f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 4} = n^2 = 0(n^2)$$

Hence, case 2 can be applied.

$$f(n) = \theta(n^{\log_b a}) = \theta(n^2)$$

Thus solution to recurrence is,

$$T(n) = \theta(n^{\log_b a} \lg n) = \theta(n^2 \lg n).$$

Example 3.30. Use iteration to solve $T(n) = T(n - a) + T(a) + n$, where $a \ge 1$ is a constant.

Solution.

$$\begin{aligned}
T(n) &= T(n-a) + T(a) + n \\
&= (T(n-2a) + T(a) + n - a) + T(a) + n \\
&= (T(n-3a) + T(a) + n - 2a) + 2T(a) + 2n - 3 \\
&= \sum_{i=0}^{n/a} T(a) + \sum_{i=0}^{n/a} n - ia \\
&= \left(\frac{n}{a}\right) T(a) + \sum_{i=0}^{n/a} n - a \sum_{i=0}^{n/a} i \\
&= \left(\frac{n}{a}\right) T(a) + n \sum_{i=0}^{n/a} 1 - a \sum_{i=0}^{n/a} i \\
&= \left(\frac{n}{a}\right) T(a) + n\left(\frac{n}{a}\right) - a\left(\frac{n}{a^2}\right)\Big/2.
\end{aligned}$$

Example 3.31. Characteristic each of the following recurrence equations using the master method (assuming that $T(n) = c$ for $n < d$, for constants $c > 0$ and $d \ge 1$).

(*a*) $T(n) = 2T(n/2) + \log n$

(*b*) $T(n) = 8T(n/2) + n^2$

(*c*) $T(n) = 16T(n/2) + (n \log n)^4$

(*d*) $T(n) = 7T(n/3) + n$

(*e*) $T(n) = 9T(n/3) + n^3 \log n$

Solution. (*a*) $T(n)$ is $O(n)$ (case 1).

(*b*) $T(n)$ is $O(n^3)$ (case 1).

(*c*) $T(n)$ is $O(n^4 \log^5 n)$ (case 2).

(*d*) $T(n)$ is $O(n^{\log_3 7})$ (case 1).
(*e*) $T(n)$ is $O(n^3 \log n)$ (case 3).

EXERCISES

1. Solve the recurrence exactly for n^a power of 2.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 5T\left(\frac{n}{2}\right) + (n \lg n)^2 & \text{if otherwise} \end{cases}$$

2. Solve the following recurrence exactly for n a power of 2.

$$T(n) = \begin{cases} \frac{1}{3} & \text{if } n = 1 \\ \frac{3}{2} & \text{if } n = 2 \\ \frac{3}{2}T\left(\frac{n}{2}\right) - \frac{1}{2}T\left(\frac{n}{4}\right) & \text{otherwise} \end{cases}$$

3. Solve the following recurrence exactly

$$T(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \sqrt{\frac{1}{2}T^2(n-1) + \frac{1}{2}T^2(n-2) + n} & \text{otherwise} \end{cases}$$

4. Solve the following recurrence for n of the form 2^{2^k}

$$T(n) = \begin{cases} 1 & \text{if } n = 2 \\ 2T(\sqrt{n}) + \lg n & \text{otherwise} \end{cases}$$

5. Solve the following recurrence enitly as a function of initial conditions a and b.

$$T(b) = \begin{cases} a & \text{if } n = 0 \\ b & \text{if } n = 1 \\ 1 + T(n-1)/T(n-2) & \text{otherwise} \end{cases}$$

6. Solve the following recurrences, where $T(1) = 1$ and $T(n)$ for $n \geq 2$ satisfies.

(*a*) $$T(n) = 3T\left(\frac{n}{2}\right) + n$$

(*b*) $$T(n) = 3T\left(\frac{n}{2}\right) + n^2$$

(*c*) $$T(n) = 8T\left(\frac{n}{2}\right) + n^3$$

(*d*) $$T(n) = 8T\left(\frac{n}{3}\right) + n$$

(*e*) $$T(n) = 4T\left(\frac{n}{3}\right) + n^2$$

(f) $$T(n) = 9T\left(\frac{n}{3}\right) + n^3.$$

7. Give tight big-oh and big-omega bounds on defined by the following recurrences. Assume $T(1) = 1$

(a) $$T(n) = T\left(\frac{n}{2}\right) + 1$$

(b) $$T(n) = 2T\left(\frac{n}{2}\right) + \lg n$$

(c) $$T(n) = 2T\left(\frac{n}{2}\right) + n$$

(d) $$T(n) = 2T\left(\frac{n}{2}\right) + n^2.$$

8. Solve for $T(n)$:

$$T(1) = 1$$

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \text{ for } n \geq 2.$$

9. Solve the following recurrence $T(1) = 0$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + n - 1.$$

10. Find the closed form expression and their asymptotic notations

(a) $\sum_{i=0}^{n} i$ (b) $\sum_{i=0}^{n} i^k$ (c) $\sum_{i=0}^{n} 2i$ (d) $\sum_{i=0}^{n} \left[\binom{n}{i}\right]$

QUESTIONS WITH ANSWERS

Q. 1. Solve the recurrence $T(n) = T\left(\frac{n}{3}\right) + n^k$, by iteration method.

Ans.

$$\begin{aligned} T(n) &= T\left(\frac{n}{3}\right) + n^k \\ &= n^k + T\left(\frac{n}{3}\right) \\ &= n^k + \left(\frac{n}{3}\right)^k + T\left(\frac{n}{9}\right) \\ &= n^k + \left(\frac{n}{3}\right)^k + \left(\frac{n}{9}\right) + \ldots + \left(\frac{n}{3^i}\right)^k \end{aligned}$$

When $\frac{n}{3^i} = 1 \Rightarrow n = 3^i \Rightarrow i = \log_3 n.$

Now for upper bound,

$$T(n) \leq n^k + n^k\left(\frac{1}{3^k}\right) + n^k\left(\frac{1}{3^k}\right)^2 + \ldots + n^k\left(\frac{1}{3^k}\right)^i$$

$$= n^k \sum_{j=0}^{i} \left(\frac{1}{3k}\right)^j$$

$$\le n^k \sum_{j=0}^{\infty} \left(\frac{1}{3k}\right)^j \le n^k \frac{1}{1-\frac{1}{3^k}} = O(n^k)$$

Hence, $T(n) = O(n^k)$.

Q. 2. Use a recurrence tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T\left(\lfloor \frac{n}{4} \rfloor\right) + cn^2$.

Ans.

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$

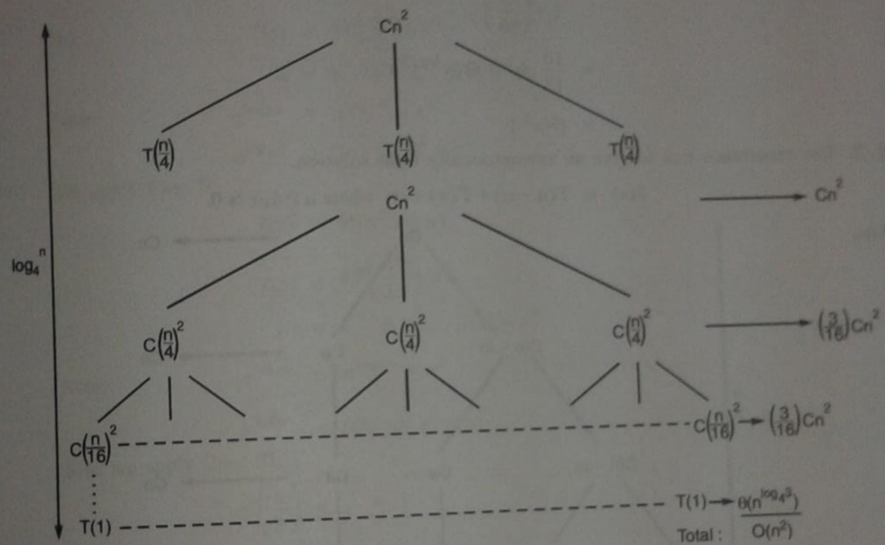


Fig. 3.5.

The sub-problem size for a node at depth i is $\frac{n}{4^i}$.

When $\frac{n}{4^i} = 1 \Rightarrow i = \log_4 n$

Now we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i, from root, each node at depth i, for $i = 0, 1, 2, ..., \log_4 n - 1$, has cost of $c\left(\frac{n}{4^i}\right)^2$.

Now total cost is $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$

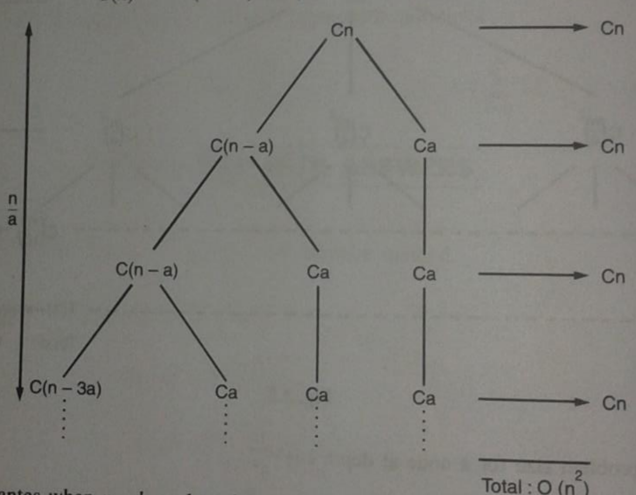
Now, we add up the costs

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2).
\end{aligned}
$$

Q. 3. Use recurrence tree to give an asymptotically tight solution.

$$T(n) = T(n-a) + T(a) + cn \text{ where } a \geq 1, c > 0.$$

Ans.



Series termiantes when $n - ka = 1 \Rightarrow ka = n - 1$

$$k = \frac{(n-1)}{a} = \frac{n}{a} \text{ levels}$$

$$\text{Total cost, } T(n) = \left(\frac{n}{a}\right) . c_n = O(n^2).$$

Q. 4. Use master method to give tight asymptotic bounds for the following recurrence.

(a) $$T(n) = 4T\left(\frac{n}{2}\right) + n$$

(b) $$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

(c) $$T(n) = 4T\left(\frac{n}{2}\right) + n^3.$$

Ans. (a) $$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$$a = 4, \quad b = 2, \quad f(n) = n$$

Now, $$n^{\log_b a} = n^{\log_2 4}$$

$$n^{\log_b a} > f(n) \text{ i.e. } n^2 > n$$

We apply Case I,

$$\therefore \quad T(n) = \Theta(n^2)$$

(b) $$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$a = 4, \quad b = 2, \quad f(n) = n^2$$

Now, $$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$n^{\log_b a} = f(n) \text{ i.e. } n^2 = n^2$$

We apply Case II,

$$\therefore \quad T(n) = \Theta(n^2 \log n).$$

(c) $$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

$$a = 4, \quad b = 2, \quad f(n) = n^3$$

Now, $$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$n^{\log_b a} < f(n) \text{ i.e. } n^2 < n^3$$

Now we apply Case III,

$$af\left(\frac{n}{b}\right) \le c \cdot f(n)$$

$$4\frac{n^3}{8} \le cn^3$$

$$c \ge \frac{1}{2}$$

Solution is $T(n) = \Theta(n^3)$.

Q. 5. Can the master method be applied to the recurrence $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$? Why or why not.

Ans. $$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$$

It has the proper form,

$$a = 4, \quad b = 2, \quad f(n) = n^2 \log n$$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

It might seem that Case III should apply, since $f(n) = n^2 \log n$ is asymptotically larger than $n^2 = n^{\log_b a}$.

The problem is that, it is not polynomially larger. The ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \log n}{n^2} = \log n$ is asymptotically less than n^ε for any positive constant ε. The recurrence falls into the gap between case II and Case III. So we can't determine. Hence master method can't apply.

Q. 6. Use the master method to show that the solution of recurrence $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$ is $T(n) = \Theta(\lg n)$.

Ans.

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$a = 1, \quad b = 2, \quad f(n) = \Theta(1) = 1 = n^0$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = n^{\log_b a} \qquad \text{i.e. } n^0 = n^0$$

We apply Case II.

$$T(n) = \Theta(\lg n)$$

Q. 7. Solve the recurrence $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$.

Ans.

$$a = 3, \quad b = 4, \quad f(n) = n \lg n$$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$$

Since, $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, where $\varepsilon \approx 0.2$ Case III applies, if we can show that the regularity condition holds for $f(n)$.

$$af\left(\frac{n}{b}\right) \le c\, f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le c\, n \log n, \qquad c = \frac{3}{4}$$

Hence, $T(n) = \Theta(n \log n)$.

Q. 8. The recurrence $T(n) = 7T\left(\frac{n}{2}\right) + n^2$ describes the running time of an algorithm A. A competing algorithm A' has a running time $T'(n) = aT'\left(\frac{n}{4}\right) + n^2$. What is the largest integer value for a such that A' is asymptotically faster than A ?

Ans. For A :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

Using Master method,

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

It might seem that Case III should apply, since $f(n) = n^2 \log n$ is asymptotically large than $n^2 = n^{\log_b a}$.

The problem is that, it is not polynomically larger. The ratio $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \log n}{n^2} = \log n$ is asymptotically less than n^{ϵ} for carry any +ve constant $\in$. The recurrence falls into the gap between case II and Case III. So we can't determine. Hence master method can't apply.

Q. 6. Use the master method to show that the solution of recurrence $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$ is $T(n) = \Theta(\lg n)$.

Ans.

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

$$a = 1, \quad b = 2, \quad f(n) = \Theta(1) = 1 = n^0$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = n^{\log_b a} \qquad i.e.\ n^0 = n^0$$

We apply Case II,

$$T(n) = \Theta(\lg n).$$

Q. 7. Solve the recurrence $T(n) = 3T\left(\frac{n}{3}\right) + n \log n$.

Ans.

$$a = 3, \quad b = 4, \quad f(n) = n \lg n$$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.90}$$

Since, $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ where $\in = 0.2$ Case III applies, if we can show that the regularity condition holds for $f(n)$.

$$af\left(\frac{n}{b}\right) \le c.f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le c\, n \log n, \qquad c = \frac{3}{4}$$

Hence, $T(n) = \Theta(n \log n)$.

Q. 8. The recurence $T(n) = 7T\left(\frac{n}{2}\right) + n^2$ describes the running time of an algorithm A. A competing algorithm A' has a running time $T'(n) = aT'\left(\frac{n}{4}\right) + n^2$. What is the largest integer value for 'a' such that A' is asymtotically faster than A ?

Ans. For A :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2$$

Using Master method,

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 7}$$

Now, $\quad n^{\log_2 7} > n^2$

We apply Case-I

$$T(n) = \Theta(n^{\log_2 7})$$

For A' : $\quad T'(n) = aT'\left(\frac{n}{4}\right) + n^2$

$$a = a, \quad b = 4, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_4 a}$$

but $f(n)$ is same in both algorithms, which lead us to try the first case. Now,

$$n^{\log_4 a} < n^{\log_2 7}$$

$$\log_4 a < \log_2 7$$

$a < 49$ which happens when $a = 48$.

A' is asymptotically faster than A as long as $a < 48$. Hence A' is asymptotically faster than A upto $a = 48$.

Q. 9. Solve $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$.

Ans. $\quad T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

Here, $\quad a = 2, \quad b = 4, f(n) = n^{1/2}$

$$n^{\log_b a} = n^{\log_4 2} = n^{\log_{(2)^2} 2} = n^{\frac{\log_2 2}{2}}$$

$$n^{\log_b a} = n^{1/2} = \sqrt{n}$$

Case II apply.

Hence, $\quad T(n) = \Theta(\sqrt{n}\log n)$.

Q. 10. Solve $T(n) = 2T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + n$.

Ans. Because we know that the floor and cell functions are usually insubstantial in solving the recurrences, we convert the given equation, into

$$T(n) = 2T\left(\frac{n}{3}\right) + n$$

$$T\left(\frac{n}{3}\right) = 2T\left(\frac{n}{3}\right) \text{ and } T\left(\frac{n}{9}\right) = 2T\left(\frac{n}{27}\right) + \frac{n}{9}$$

$$T(n) = 2T\left(\frac{n}{3}\right) + n$$

$$= 2\left[2T\left(\frac{n}{9}\right) + \frac{n}{3}\right] + n = 2^2\, T\left(\frac{n}{9}\right) + \frac{2n}{3} + n$$

$$= 2^2\left[2T\left(\frac{n}{27}\right)+\frac{n}{9}\right]+\frac{2n}{3}+n$$

$$= 2^3T\left(\frac{n}{27}\right)+\left(\frac{2}{3}\right)^2 n+\left(\frac{2}{3}\right)n+\left(\frac{2}{3}\right)^0 n$$

Suppose after i iteration we approached to the boundary condition of size $n = 1$.

$$T(n) = 2^iT\left(\frac{n}{3^i}\right)+\left(\frac{2}{3}\right)^{i-1} n+\left(\frac{2}{3}\right)^{i-2} n+\ldots+\left(\frac{2}{3}\right)^0 n$$

Here $\frac{n}{3^i} \le 1 \Rightarrow 3i \ge n \Rightarrow i \ge \log_3 n$.

Substituting this value of i into above equation

$$T(n) = 2^{\log_3 n}T(1)+n\sum_{i=0}^{\log_3 n-1}\left(\frac{2}{3}\right)^i$$

$$= n^{\log_3 2}T(1)+\sum_{i=0}^{\log_3 n-1}\left(\frac{2}{3}\right)^i$$

$$\le n^{\log_3 2}\Theta(1)+n\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^i = 0(n)+n\frac{1}{1-\frac{2}{3}} = 0(n)+3$$

$T(n) = 0(n)$. **Ans.**

Q. 11. Show that the solution of the recurrence $T(n) = 3T\left(\left[\frac{n}{4}\right]\right)+n$ is $0(n)$.

Ans. Because we know that the floor and cell functions are usually insubstantial while solving recurrence relations. So omitting floor function we get

$$T(n) = 3T\left(\frac{n}{4}\right)+n$$

$$= 3\left[3T\left(\frac{n}{16}\right)+\frac{3}{4}\right]+n$$

$$= 3^2T\left(\frac{n}{16}\right)+\frac{3}{4}n+n$$

$$= 3^2\left[3T\left(\frac{n}{64}\right)+\frac{n}{16}\right]+\frac{3}{4}n+n$$

$$= \ldots\ldots\ldots\ldots\ldots\ldots$$

$$= 3^iT\left(\frac{n}{4^i}\right)+\left(\frac{3}{4}\right)^{i-1} n+\left(\frac{3}{4}\right)^{i-2} n+\ldots\left(\frac{3}{4}\right)^0 n$$

Suppose after i iterations we approached to the initial condition

i.e. $\frac{n}{4^i} \le 1 \Rightarrow 4^i \le n \Rightarrow i \le \log_4 n$

Substituting the value of i in the above equation we get

$$T(n) = O(n)$$

Q. 12. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

Ans.

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

$$= 2T\left(\frac{n}{2}\right) + n \log^{-1} n = 2T\left(\frac{n}{2}\right) - n \log n$$

Applying the master method

$$a = 2, b = 2, f(n) = -n \log n, n^{\log_b a} = n^{\log_2 2} = n' = n$$

Here $f(n)$ is larger than $n^{\log_b a}$, but it is not polynomial lager, while it is simply logarithmically larger only. So that is why we can't apply master theorem in this recurrence.

OBJECTIVE TYPE QUESTIONS

1. The time complexity of the following function is (assume $n > 0$)

```
int recursive (int n)
{  if (n == 1) return(1);
   else return (recursive (n - 1) + recursive (n - 1));
}
```

(*a*) $O(n)$ (*b*) $O(n \log n)$

(*c*) $O(n^2)$ (*d*) $O(2^n)$

2. The following recurrence equation evaluates to

$$T(1) = 1 \quad \text{for } n = 1$$

$$T(n) = 2T(n-1) + n \quad \text{for } n \le 2$$

(*a*) $2n + 1 - n - 2$ (*b*) $2n - n$

(*c*) $2n + 1 - 2n - 2$ (*d*) $2n + n$

3. Suppose $T(n) = 2T(n/2) + n$, $T(0) = T(1) = 1$. Which one of the following is false ?

(*a*) $T(n) = O(n^2)$ (*b*) $T(n) = Q(n \log n)$

(*c*) $T(n) = \Omega(n^2)$ (*d*) $T(n) = O(n \log n)$

4. What is the time complexity of the following recursive algorithm ?

```
int doSomething(int n)
{  if (n <= 2)
         return 1;
     else
         return (doSomething(floor(sqr(n))) + n);
}
```

(*a*) $Q(n^2)$ (*b*) $Q(n \log_2 n)$

(*c*) $Q(\log_2 n)$ (*d*) $Q(\log_2 \log_2 n)$

ANSWERS

1. (*d*)	**2.** (*a*)	**3.** (*b*)	**4.** (*d*)

Chapter 4

Divide and Conquer

In computer science, divide and conquer is an imporant algorithm design paradigm. It works by recursively breaking down a problem into two or more sub-problems of the same type, until these becomes simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. A divide and conquer algorithm is closely tied to a type of recurrence relation between functions of the data in questions; data is "divided" into smaller portions and the result calculated thence. The divide-and-conquer algorithms consists of three steps :

1. Breaking the problem into several sub-problems that are similar to the original problem but smaller in size.
2. Solve the sub-problem recursively (successively and independently) and then
3. Combine these solutions to subproblems to create a solution to the original problem.

The technique named "divided-and-conquer" because a problem is conquered by divided it into several smaller problems. This technique yields elegant, simple and quite often very efficient algorithms.

For example, if the work of splitting the problems and combining the partial solutions is proportional to the problems's size n, there are a bounded number b of sub-problems of size $\frac{n}{b}$ at each stage, and the base cases requires $O(1)$ time, then the divide-and-conquer algorithm will have $O(n \log n)$ complexity.

Divide and conquer algorithms typically have a time complexity recurrence relation of the form :

$$T(1) = c$$

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k$$

Such that a problem of size n is split into "a" subproblems, each of size $\frac{n}{b}$. And cn^k is the cost for this recurrecen depends on the relation between a and b^k (n is power of b : *i.e.* $n = b^k$)

If $a > b^k$, then $T(n) = O(n^{\log_b a})$

If $a = b^k$, then $T(n) = O(n^k \log n)$

If $a < b^k$, then $T(n) = O(n^k)$

4.1. MULTIPLICATION OF TWO n-BIT NUMBERS

The traditional method of multiplication of two n-bit numbers requires $O(n^2)$ bit operations. The divide-and-conquer method provides a solution of complexity $O(n^{\log 3}) = O(n^{1.59})$. Let X and Y be two n-bit numbers. Assume n is a power of 2. We partition X and Y each into two halves as shown below. Each half is of $n/2$ bits.

X	A	B
Y	C	D

We may write the product as :

$$\begin{aligned} Z &= XY = (A2^{n/2} + B)(C2^{n/2} + D) \\ &= AC2^n + (AD + BC)\,2^{n/2} + BD \end{aligned}$$

This requires 4 multiplications of $\frac{n}{2}$ bits number plus some additions and shifts (multiplication by power of 2). This way of dividing the problem and carrying out 4 multiplications will lead to $O(n^2)$ complexity. Let us try to find a faster way by reducing the number of multiplications of $\frac{n}{2}$ bit number through rearrangement in the computation process.

The product Z of X and Y can also be computed by the following program.

Algorithm 4.1. Multiply (A, B, C, D) ....

(1) $U = (A + B)$ and $(C + D)$;
(2) $V = A * C$;
(3) $W = B * D$;
(4) $Z = V * 2^n + [U - V - W] * 2^{n/2} + W$;

We assume that $A + B$ and $C + D$ have only $\frac{n}{2}$ bits, ignoring the fact that due to carry, $A + B$ and $C + D$ may be $\left(\frac{n}{2}+1\right)$ bit numbers. The scheme requires only three multiplication of $\frac{n}{2}$ bits, numbers, plus some additions and shifts, to multiply two n-bit numbers. One can use the multiplication routine recursively to evaluate the product U, V and W. The addition and shifts require time $O(n)$. Thus the time complexity of multiplying two n-bit numbers is bounded from above by :

$$\begin{aligned} T(n) &= k && \text{for } n = 1 \\ &= 3T\left(\frac{n}{2}\right) + kn, && \text{for } n > 1 \end{aligned}$$

where k is a constant reflecting the costs of additions and shifts.

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{2}\right) + kn \\ &= 3\left[3T\left(\frac{n}{4}\right) + k\frac{n}{2}\right] + kn \\ &= 3^2\, T\left[\left(\frac{n}{4}\right)\right] + 3\frac{kn}{2} + kn \\ &= 3^3\, T\left(\frac{n}{2^3}\right) + \left(\frac{3}{2}\right)^2 + kn + kn \end{aligned}$$

Let $n = 2^P$, then we may write $T(n)$ as

$$\begin{aligned} T(n) &= 3^P\, T(1) + \left(\frac{3}{2}\right)^{P-1} kn + \ldots + \left(\frac{3}{2}\right) kn + kn \\ &= k3^P + kn\left[\left(\frac{3}{2}\right)^{P-1} + \ldots + \left(\frac{3}{2}\right) + 1\right] \\ &= k3^P + 2kn\left[\left(\frac{3}{2}\right)^P - 1\right] \\ &= k3^{\log_2 n} + 2kn\left[\left(\frac{3^{\log_2 n}}{2^{\log_2 n}}\right) - 1\right] \\ &= k3^{\log_2 n} + 2kn\left[\left(\frac{3^{\log_2 n}}{n}\right) - 1\right] \\ &= 3kn^{\log_2 3} - 2kn \quad \left(\text{Since } 3^{\log_2 n} = n^{\log_2 3}\right) \end{aligned}$$

Therefore, $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$

Suppose

$$\begin{aligned} X &= 0101,\ Y = 0110.\ \text{Then } A = 01,\ B = 01,\ C = 01,\ D = 01 \\ U &= (A + B)\ (C + D) = 110,\ V = AC = 001,\ W = BD = 010 \\ W &= V * 2^4 + (U - V - W) * 2^2 + W \\ &= 00010000 + 1100 + 010 = 00011110 \end{aligned}$$

4.2. BINARY SEARCH

A binary search algorithm is a technique for finding a particular value in a sorted list. It makes progressively better guesses, and closes in on the sought value by selecting the median element in a list, comparing its value to the target value (key), and determining if the selected value is greater than, less than, or equal to the target value. A guess that turns out to be too high becomes the new top of the list, and a guess that is too low becomes the new bottom of the list. Pursuing this

approach iteratively, it narrows the search by a factor of two each time, and finds the target value.
The binary search consists of the following steps :

1. Search a sorted array by repeatedly dividing the search interval in half.
2. Begin with an interval covering the whole array.
3. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half.
4. Repeatedly check until the value is found or the interval is empty.

The most straightforward implementation is recursion, which recursively searches the sub-array dictated by the comparison :

Algorithm 4.2. Binary Search (arr [1..*n*], value, low, high) ....

```
If (high < low)
    return – 1   // not found
Endif
mid = (low + high) / 2
If (arr[mid] > value)
    return BinarySearch(arr, value, low, mid-1)
Else If (arr[mid] < value)
        return BinarySearch(arr, value, mid + 1, high)
    Else
            return mid  // found
        Endif
Endif
```

The algorithm is invoked with initial low and high values of 1 and *n*. We can eliminate the recursion above and convert this to an iterative implementation. Iterative one is presented in Algorithm 1.11. Figure 4.1 illustrates trace of the algorithm to find the target value of 65.

```
mid = (low + high)/2 = (1 + 9)/2 = 5
arr[mid] < value    (55 < 65)
low = mid + 1 = 5 + 1 = 6
high = 9

mid = (low + high)/2 = (6 + 9)/2 = 7
arr[7] > value  (75 > 65)
high = mid - 1 = 7 - 1 = 6
mid = (low + high)/2 = (6 + 6)/2 = 6
arr[6] > value  (65 > 65) → false
arr[6] < value  (65 < 65) → false
arr[6] = value  (65 = 65) → false
Found at mid = 6.
```

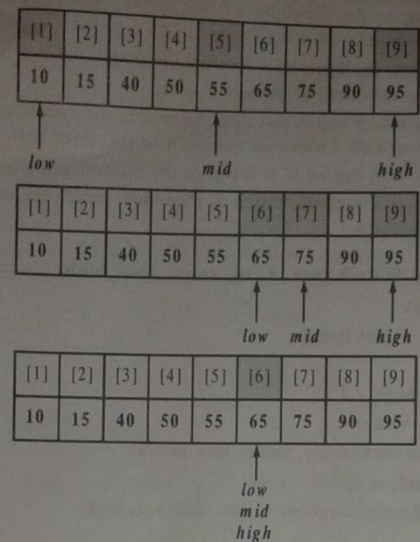


Fig. 4.1. Trace of binary search.

Binary is a logarithmic algorithm and runs in $O(\log_2 n)$ time. Specifically, $1 + \log_2 n$ iteration are needed to return an answer. In most cases it is considerably faster than a linear search. It can be implemented using recursion or iteration, as shown above.

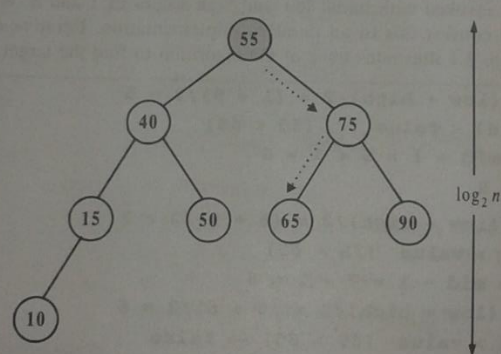


Fig. 4.2. Trace of binary search algorithm to find 65.

Let us verify the height (or depth) of the binary search tree with $\log_2 n$ values. Notices the following table. Binary search tree of Fig. 4.2 has $n = 9$ nodes. Hence, its height is $\lfloor \log_2 9 \rfloor = 3$. Note that this is an upper bound.

TABLE 4.1. $\log_2 n$ Values for Different Sizes of Binary Search Tree

n	1	2	3	4	5	6	7	8	9
$\log_2 n$	0.00	1.00	1.58	2.00	2.32	2.58	2.81	3.00	3.17
$\lfloor \log_2 n \rfloor$	0	1	1	2	2	2	2	3	3

n	15	16	17	31	32	33	64	65	128
$\log_2 n$	3.91	4.00	4.09	4.95	5.00	5.04	6.00	6.02	7.00
$\lfloor \log_2 n \rfloor$	3	4	4	4	5	5	6	6	7

4.2.1. Analysis of Binary Search

The time complexity for the binary search may be written as :

$$T(n) = \begin{cases} k, & n = 1 \\ T\left(\frac{n}{2}\right) + k, & n > 1 \end{cases}, \text{ where } k \text{ is any constant}$$

$$T(n) = T\left(\frac{n}{2}\right) + k$$

$$= T\left(\frac{n}{2^2}\right) + 2k = T\left(\frac{n}{2^3}\right) + 3k$$

....

$$T(n) = T\left(\frac{n}{2^r}\right) + rk$$

$$= T(1) + rk, \text{ where } 2^r = n, r \text{ is a positive integer.}$$

$$= (r + 1)\, k$$

$$= (1 + \log_2 n)\, k = O(\log n)$$

4.3. MERGE SORT

Merge sort is an $O(n \log n)$ comparison-based sorting algorithm. In most implementations it is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm.

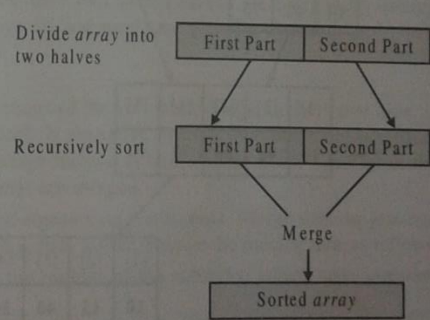


Fig. 4.3.

Sorting by merging is a recursive, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of elements $(n > 1)$:

(*i*) Divide the sequence into two sequences of length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$

(*ii*) Recursively sort each of the two subsequences; and then

(*iii*) Merge the sorted subsequences to obtain the final result.

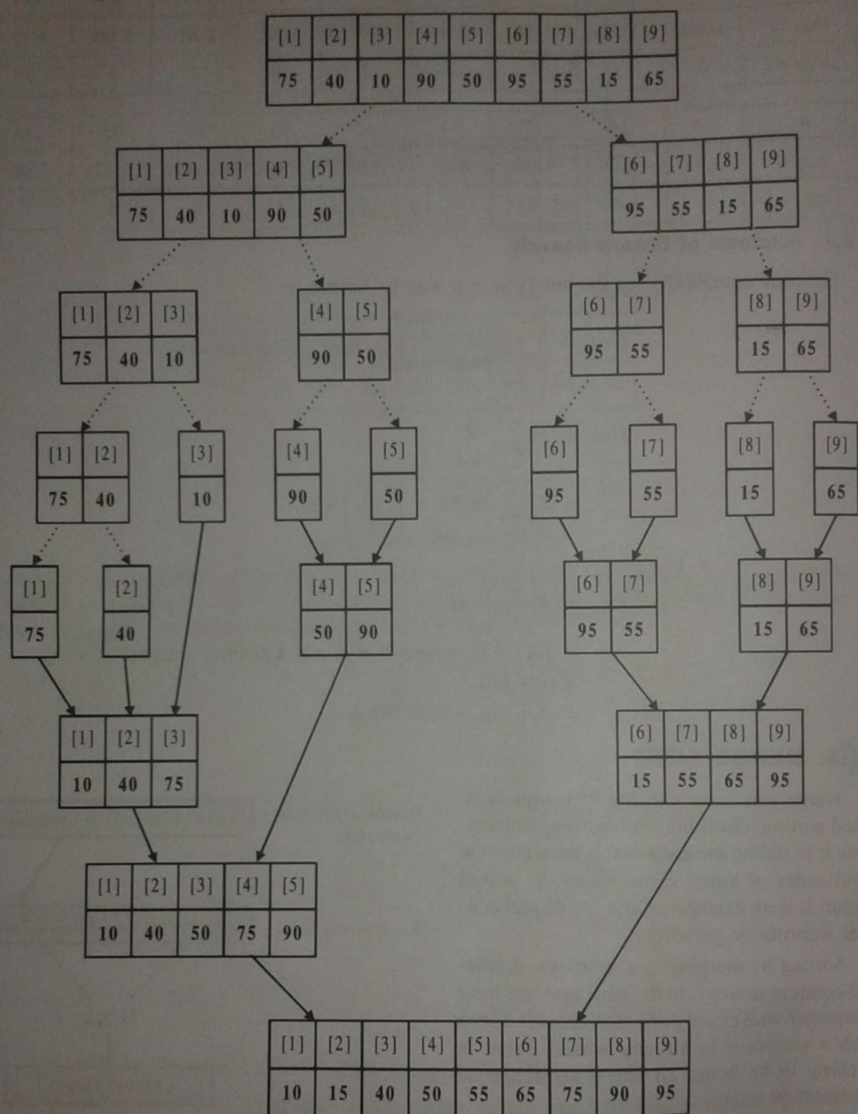


Fig. 4.4. Two-way merge sort

Figure 4.3 shows the idea of merge sort. Fig. 4.4 illustrates the operation of the two-way merge sort algorithm. We assume to sort the given array $a[n]$ into ascending order. We split it into two subarrays : $a[1]....a[\lceil n/2 \rceil]$ and $a[\lceil n/2 \rceil + 1] ... a[n]$. Each subarray is individually sorted, and the resulting sorted subarrays are merged to produce a single sorted array of n elements. Consider an array of nine elements : {75, 40, 10, 90, 50, 95, 55, 15, 65}. The merge() algorithm divides the array into subarrays and merges them into sorted subarrays by merge() algorithm as illustrated in the Fig. 4.4 (Dashed-line arrows indicate the process of splitting and the regular arrows the merging process).

First, p half of the array with five elements is being split and merged; and next second half of the array with four elements is processed (Notice how the splitting continues until subarrays containing a single element are produced). Fig. 4.4 represents the sequence of recursive calls that are produced by *mergeSort* when it is applied to nine elements.

Since merge operations cannot be done in place, a second temporary array, tmp[] is needed, in addition to the main array, a[].

The *merge*( ) algorithm takes three integer parameters : p, q (middle), and r. The relationship among these parameters is :

$$p \le q < r$$

It is assumed that the two subsequences of the array, a [] :

```
a[p], a[p + 1], ..., a[q],
```

and

```
a[q + 1], a[q + 2], ..., a[r],
```

are both sorted. The merge() method merges the two sorted subarrays using the temporary array, tmp[]. It then copies the merged and sorted sequence into the array at

```
a[p], a[p + 1], ...., a[r]
```

The mergesort() algorithm implements the recursive divide-and-conquer by taking two parameters: p and r (Initially, $p = 1$, $r = n$). These parameters specify the two subsequences of the array, a[] to be sorted. If the sequence to be sorted contains more than one element, the sequence is split in two, each half is recursively sorted, and then two sorted halves are merged by calling merge() algorithm. We assume that the two arrays a[] and tmp[] are globally declared.

4.3.1. Analysis of Merge Sort

In merge algorithm, the total number of iterations of the two while-loops, in the worst case, is $r - p$. The number of iterations of the third for loop is the same. Since all the loops do a constant amount of work, the total running time for the merge method is $O(n)$, where $n = r - p$. This is the total number of elements in the two sub-arrays that are merged.

We analyze the worst-case running time of mergesort on n elements. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

(*i*) **Divide :** The divide step just computes the middle of the subarray, which takes constant time, $O(1)$.

(*ii*) **Conquer :** We recursively solve two subproblems, each of size $n/2$, which contributes $T(n/2) + T(n/2)$ to the running time.

(*iii*) **Combine :** The merge procedure on an n-element subarray takes time $O(n)$.

We add a function that is $O(n)$ and a function that is $O(1)$. This sum is a linear function of n, that is, $O(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of mergesort :

$$\boxed{\begin{aligned} T(1) &= O(1) && \text{for } n = 1 \\ T(n) &= 2T(n/2) + O(n) && \text{for } n > 1 \end{aligned}}$$

This can be solved by repeated substitutions. Finally,

$$T(n) = n \log_2 n + n$$

Therefore, the running time of merge sort is $O(n \log_2 n)$.

Algorithm 4.3. mergesort (p, r) ....

```
If (p < r)
    q = (p + r)/2            // Divide problem into subproblems
    mergesort (p, q)         // Solve first subproblem
    mergesort (q + 1, r)     // Solve second subproblem
    merge (p, q, r)          // Combine the solutions
Endif
```

Algorithm 4.4. merge (p, q, r) ....

```
i = j = p; k = q + 1
While j ≤ q and k ≤ r
    If a[j] < a[k]
        tmp[i] = a[j]; Increment i and j
    Else
        tmp[i] = a[k]; Increment i and k
    Endif
Endwhile
While j ≤ q
    tmp[i] = a[j]; Increment i and j
Endwhile
For i = p to k
    a[i] = tmp[i]
Endfor
```

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by :

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T\left(\dfrac{n}{2}\right) + Cn, & n > 1 \end{cases} \qquad [C \text{ is a constant}]$$

where n is a power of 2, *i.e.* $n = 2^k$, we proceed as follows :

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + Cn \\
&= 2\left(2T\left(\frac{n}{4}\right) + C\frac{n}{2}\right) + Cn \\
&= 4T\left(\frac{n}{4}\right) + 2\,Cn \\
&= 8T\left(\frac{n}{8}\right) + 3\,Cn \\
&= 2^k\,T\left(\frac{n}{2^k}\right) + k\,Cn \\
&= 2^k\,T(1) + k\,Cn \\
&= k\,Cn = Cn \log_2 n
\end{aligned}
$$

It is easy to see that if $2^k < n < 2^{k+1}$, then

$$T(n) \le T(2^{k+1})$$

therefore, $$T(n) = O(n \log n)$$

4.4. QUICK SORT

As its name implies, quicksort is the fastest known sorting algorithm in practice. The algorithm solely depends on the data it receives. If the data has certain properties quicksort is one of the fastest, if not, quicksort can be very slow. Quicksort can perform quite fast, on average about $O(n \log n)$, but its worst case is a degrading $O(n^2)$. For quicksort, the worst case is usually when the data is already sorted.

Quicksort sorts by using a divide and conquer strategy to divide a list into two sub-lists. Quicksort is naturally recursive. We partition the array into two sub-arrays, and then restart the algorithm on each of these sub-arrays. The partition procedure involves choosing some object (usually, already in the array - pivot); If some other object is greater than the chosen object, it is added to one of the sub-arrays, if it is less than the chosen object, it is added to another sub-array. Thus, the entire is partitioned into two sub-arrays, with one sub-array having enverything that is larger than the chosen object, and the other sub-array having everything that is smaller. A general algorithm based on a recursive method follows :

Variable a is an integer array of size, n. The p and r invoke procedure and they are initialized with 1 and n respectively; and are the current lower and upper bounds of the sub-arrays. The indices newleft and new$r$ are used to select certain elements during the processing of each sub-array. Variable amid (pivot) is the element which is placed in its final location in the array.

Algorithm 4.5. Quicksort (a, *p*, *r*)..

1. newleft = p; newright = r.
2. amid = a [(p + r)/2] // pivot
3. While newleft ≤ newright // Partition
4. While (a[newleft] < amid and newleft < r)// while loop-1
5. Increment newleft

```
Endwhile
While (amid < a[newright] and newright > p)// while loop-2
   Decrement newright
Endwhile
   If newleft ≤ newright, then
      Swap (a[newleft], a [newright])
      Increment newleft and Decrement newright
   Endif
Endwhile   // Partition ends
If p < newright, then
   Call Quicksort(a, left, newright) // sort left sub-array
Endif
If newleft < r, then
   Call Quicksort (a, newleft, right) // sort right sub-array
Endif
```

Index p scans the list from p to r, and index r scans the list from r to p. A swap is performed when p is at an element larger than the pivot and r is at an element smaller than the pivot. A final swap with the pivot completes the divide step. The pivot element is placed in its final proper position in the array. We take the pivot as the middle element of the array (or sub-array). We consider an unsorted array of size 12 to sort the array in ascending order : {65, 35, 15, 90, 75, 45, 40, 60, 95, 25, 85, 55}.

$p = 1$, $r = 12$, newleft = p, newright = r, mid = (newleft + newwright)/2 = 6, amid = a[mid] = a[6] = 45. The middle element of the array is 45 which acts as a pivot. Now, p sublist is 1 to 5 and r sublist is 7 to 12. The while loop-1 scans p sublist from p to r as the element 65 is greater than pivot (45), 65 is to be shifted to r sublist. The newlight = 1. The while loop-2 scans r sublist from r to p as the element 25 is less than pivot (45). The newright = 10. The elements in the newleft and newright are interchanged.

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	*newleft*	*newright*
65	35	15	90	75	**45**	40	60	95	25	85	55	1	10

Fig. 4.5.

The subsequent steps are illustrated as follows (complete trace of the algorithm is given in Table 4.2.

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	*newleft*	*newright*
25	35	15	90	75	**45**	40	60	95	65	85	55	4	7

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	*newleft*	*newright*
25	35	15	40	75	**45**	90	60	95	65	85	55	5	6

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	*newleft*	*newright*
25	35	15	40	**45**	75	90	60	95	65	85	55	6	5

As $p <$ newright $(1 < 5)$, the following recursive call is made to sort the p sublist :

```
Quicksort (a, p, newright).
```

Table 4.2. Trace of Quick Sort.

Step	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	*left*	*right*	*pivot*
0	65	35	15	90	75	**45**	40	60	95	25	85	55	1	12	45
1	25	35	**15**	40	*45*	75	90	60	95	65	85	55	1	5	15
2	*15*	35	**25**	40	*45*	75	90	60	95	65	85	55	2	5	25
3	*15*	*25*	35	**40**	*45*	75	90	60	95	65	85	55	3	5	40
4	*15*	*25*	*35*	*40*	*45*	75	90	60	**95**	65	*15*	55	6	12	95
5	*15*	*25*	*35*	*40*	*45*	75	90	**60**	55	65	85	*95*	6	11	60
6	*15*	*25*	*35*	*40*	*45*	**55**	*60*	95	75	65	85	*95*	6	7	55
7	15	25	35	40	45	55	60	90	**75**	65	85	95	8	11	75
8	15	25	35	40	45	55	60	65	75	**90**	85	95	10	11	90
9	15	25	35	40	45	55	60	65	75	85	90	95	sorted array		

The best case behaviour of the quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length. In order to sort n elements, in this case the running time is in $O(n \log_2 n)$. This is because the recursion depth is $\log_2 (n)$ and on each level there are n elements to be treated (Fig. 4.6(a)).

The worst case occurs when in each recursion step an unbalanced partitioning is produced namely that one part consists of only one element and the other part consists of the rest of the elements (Fig. 4.6(c)). Then the recursion depth is $n - 1$ and quicksort runs in time $O(n^2)$. In the average case a partioning as shown in Fig. 4.6(b) is to be expected. Both the best case and the average case are the same that is, $O(n \log_2 n)$.

The choice of the comparison element (amid) determines which partition is achieved. Suppose that the first element of the sequence is chosen as comparison element. This would lead to the

worst-case behaviour of the algorithm when the sequence is initially sorted. Therefore, it is better to choose the element in the middle of the sequence as comparison element.

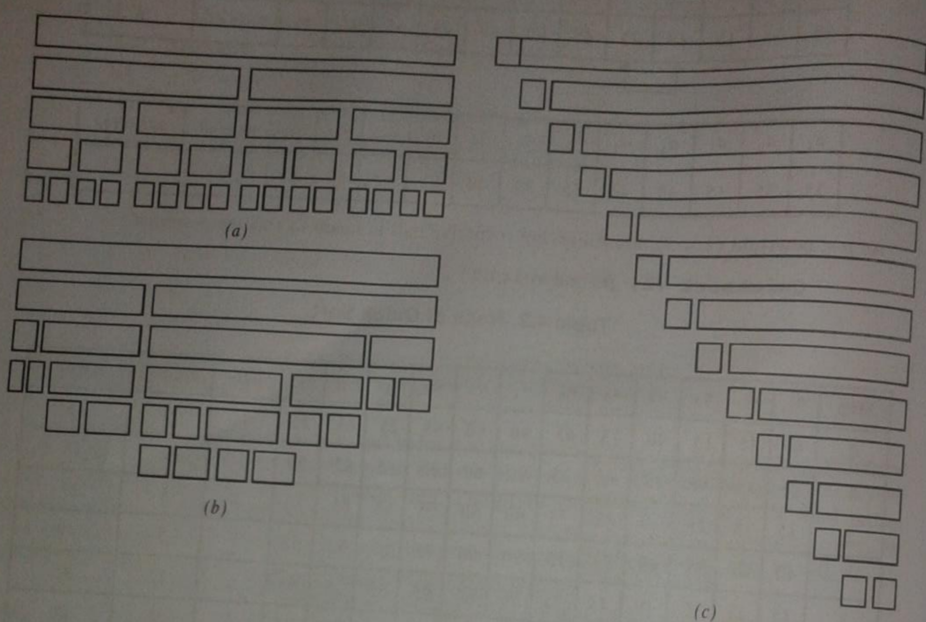


Fig. 4.6. Recursion depth of quicksort (*a*) best case, (*b*) average case, (*c*) worst case.

Even better would it be to take the $\frac{n}{2}$th greatest element of the sequence (the median). Then the optimal partition is achieved. Actually, it is possible to compute the median in linear time. This variant of quicksort would run in time $O(n \log_2 n)$ even in the worst case.

However, the beauty of quicksort lies in its simplicity. And it turns out that even in its simple form quicksort runs in $O(n \log_2 n)$ on the average. Moreover, the constant hidden in the O-notation is small. Therefore, we trade this for the (rare) worst-case behaviour of $O(n^2)$.

4.4.1. Balanced Partitioning

In average-case analysis, the array is partitioned by choosing any random number. The average-case running-time of quicksort is much closer to the best-case than to the worst-case. In this case, at each level some of the partitions are well balanced, while some are fairly unbalanced let us assume that the partition of array to be 9 : 1, then recurrence so obtained is

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n$$

$$T(n) = \Theta(n \log n)$$

$$= Cn + Cn + 4T\left(\frac{n}{4}\right)$$

$$\vdots$$

$$= Cn + Cn + \ldots + 2^i T\left(\frac{n}{2^i}\right)$$

Now, $$\frac{n}{2^i} = 1 \Rightarrow i = \log_2 n$$

$$T(n) = Cn \sum_{i=0}^{\log_2 n} 1 = Cn \log n$$

$$T(n) = O(n \lg n)$$

4.4.4. Randomized Quick Sort

We have seen that if we assume all the permutations of the number to be equally likely when selecting the "divider" $T(n) = \Theta(n \log n)$, we get the average case. We have to enforce the permutations, if we cannot assume all permutations to be equally likely.

In Randomized version all permutations of the number are not necessarily equally alike depending upon the random-number-generator, Random (a, b). Randomized version will show worst-case or average-case. Random (a, b) returns an integer x, $a \le x \le b$. It is implemented with a pseudorandom generator.

The randomized quicksort algorithm uses random (left, right) so as to obtain a new pivot element.

Algorithm 4.6 : Randomize-Partition (a, Left, Right) ....

1. Call Random (a, b) to obtain new pivot.
   set i → Random (left, right)
2. Interchange value
   set a[left] → a[i]
3. return value at the point of call
   return quicksort (a, left, right).

NOTE

Quicksort (a, left, right) is an algorithm already defined as Algorithm 4.5.

4.5. MAXIMUM AND MINIMUM

We are given a set of 'n' elements and we have to find the two element. From the given set. One is the minimum element (smallest element) and the other being the maximum element (or largest element).

To find either minimum or maximum element from the set of 'n' elements we have to do $(n - 1)$ number of comparisons.

Algorithm 4.7. Minimum (A)

```
min ← A[1]
For i ← 2 to length [A]
    do if min > A[i]
        then min ← A[i]
return min
```

To find minimum and maximum elements simultaneous. We can use a divide and conquer approach as shown below using an algorithm

Algorithm 4.8. MIN-MAX (A, p, r, min, max)

```
if (p = r) then
    max <-- A[p]
    min <-- A[p]
else if (p = r – 1) then
    if (A[p] > A[r]) then
        max <-- A[p]
        min <-- A[r]
    else max <-- A[r]
    min <-- A[p]
else mid <-- ⌊(p + r)/2⌋
    MIN-MAX (A, p, mid, min, max)
    MIN-MAX (A, mid + 1, r, min1, max1)
    if (min > min1) then
        min <-- min1
    if (max1 > max) then
        max <-- max1
Exit
```

4.5.1. Analysis of MAX-MIN

Total Time Complexity,

$$T(n) = D(n) + R(n) + C(n)$$

$$D(n) = \text{Divide time} = O(1) \text{ constant time}$$

$$R(n) = \text{Recursion time}$$

$$= 2T\left(\frac{n}{2}\right) \text{ as each time a list of } n \text{ elements are divided into two sublists of equal size.}$$

$$C(n) = \text{Combine time} = O(2) \text{ as two comparisons are required to find min and max element.}$$

Here,
$$T(n) = O(1) + 2T\left(\frac{n}{2}\right) + O(n)$$
$$= T\left(\frac{n}{2}\right) + 2$$
$$= \frac{3n}{2}$$
$$= 2n - 2$$

The no. of comparisons to find minimum and maximum value in an array is $(2n - 2)$ as there are maximum of $(n - 1)$ no. of comparisons for each operations.

If no. of elements is odd in an array, we will assume, that the first element of an array is a minimum as well as maximum valued element, therefore the time complexity of the algorithm changes to

$$T(n) = \frac{3(n-1)}{2} \leq \frac{3n}{2}$$

If number of elements in an array is even, then there will an initial comparison between the first two elements to find minimum and maximum valued elements. Hence

$$T(n) = 1 + \frac{3(n-2)}{2} = \frac{3n}{2} - 2 \leq \frac{3n}{2}$$

4.6. SUBSET SUM PROBLEM

An instance of the subset sum problem is a pair (S, t) where S is a set $\{x_1, x_2, ..., x_n\}$ of positive integers and t is a positive integer. This decision problem asks whether there exists a subset of 'S' that adds up exactly to the target value t.

We can solve this problem using divide and conquer approach but actually it is a NP-complete problem.

For example
$$S = \{1, 2, 3, 4\}$$
$$t = 5$$

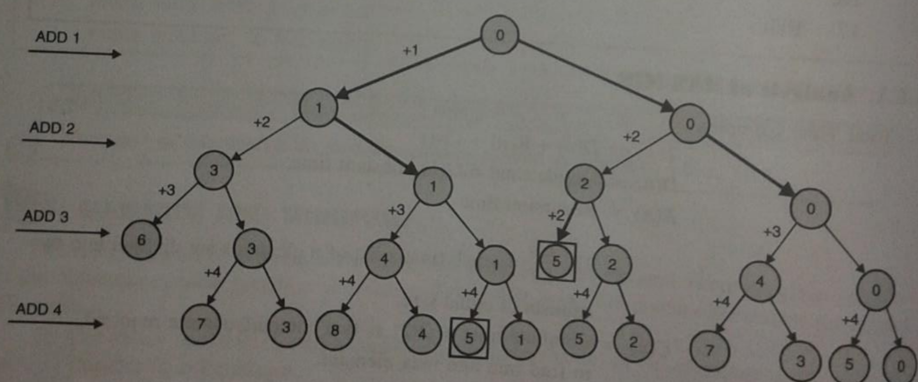


Fig. 4.8.

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

The solution of this recurrence is $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

Algorithm 4.10. Matmul (A, B, C, n)

```
1. If n = 1, then  // base case
2.    C = C + A * B
3. Else
4.    matmul (A, B, C, n/4)
5.    matmul (A, B + (n/4), C + (n/4), n/4)
6.    matmul (A + 2 * (n/4), B, C + 2 * (n/4), n/4)
7.    matmul (A + 2 * (n/4), B + (n/4), C + 3 * (n/4), n/4)
8.    matmul (A + (n/4), B + 2 * (n/4), C, n/4)
9.    matmul (A + (n/4), B + 3 * (n/4), C + (n/4), n/4)
10.   matmul (A + 3 * (n/4), B + 2 * (n/4), C + 2 * (n/4), n/4)
11.   matmul (A + 3 * (n/4), B + 3 * (n/4), C + 3 * (n/4), n/4)
12. Endif
```

EXERCISES

1. Modify the normal binary search algorithm so that in case of unsuccessful search it returns the index i such that $A[i] <$ key $< A[i + 1]$.
2. Design a divide and conquer algorithm to find the maximum and minimum of an array A of n elements, and prove that the algorithm makes at most $3n/2$ element to element comparisons.
3. What is stable sorting method ? Is mergesort a stable sorting method ?
4. Determine the running time of mergesort for
   (*i*) sorted input
   (*ii*) reverse ordered input
   (*iii*) random ordered input
5. Trace the Quicksort algorithm to sort the list C, O, L, L, E, G, E in alphabetical order. Give an instance, where the Quicksort algorithm has worst case complexity.

QUESTIONS WITH ANSWERS

Q. 1. Write and explain the control abstraction for divide and conquer.

Ans. Control abstraction for divide and conquer : Divide and conquer is a top-down technique for designing algorithms. These algorithms perform the following :

(1) Divide the problem into several subproblems that are similar to the original problem but smaller in size.

(2) Solve the subproblem (recursively, successively and independently).

(3) Then combine these solutions to subproblems to create a solution to the original problem.

Control Abstraction is an algorithm like structure. It is a procedure whose flow is clear but whose primary operations are specified elsewhere and hence unclear. Let us use a control abstraction to elaborate the idea behind the divide and conquer (dc) strategy

```
dc(P)
{  If  (P is small)
        return SimpleFunc(P);   //Simple function that can solve P
   Else
   {  Divide P into smaller instances P1, P2, ...., Pk for k ≥ 1;
      Apply dc( ) to each instance;
      Return comb_instances (dc(P1), dc(P2), ...., dc(Pk));
   }
}
```

Q. 2. Modify the normal binary search algorithm so that in case of unsuccessful search it returns the index i such that $A[i] <$ key $< A[i + 1]$.

Ans. The following is a modified binary search algorithm – if the search is successful, it returns mid, otherwise high. The index high (= i) satisfies the condition $A[i] <$ key $< A[i + 1]$ for unsuccessful search. For unsuccessful case, the condition is : low > high – that is, A[high] < key < A[log].

BinarySearch (A[1...n], key)

```
low 1; high = n;
While (low ≤ high)
    mid = (low + high) / 2;
    If (A[mid] > key), then
        high = mid – 1;
    else If (A[mid] < key), then
        low = mid + 1;
    else
        return mid;  // found
    Endif
Endwhile
return high; // not found
```

Q. 3. Devise a binary search algorithm which splits the set not into two sets of equal sizes but into two sets one-third and two-thirds. How does this algorithm compare with binary search ?

Ans. TernarySearch (a[1...n], n, x, low, high)

```
low = 1;
high = n;
while (low ≤ high)
    p = (low + high) / 3;
    If (p < low), then
        p = low;
```

```
7.      endif
8.      If (a[p] > x), then
9.          high = p – 1;
10.     else If (a[p] < x), then
11.         low = p + 1;
12.     else
13.         return p; // found
14.     Endif
15. endwhile
16. return – 1;   // not found
```

When the pointer p goes outside the range ($p <$ low), the statements 5 and 6 adjust the pointer p (p = low). The worst-case running time, T(n) of this algorithm occurs when the recursive call is on the larger $2n/3$-element partition.

Q. 4. Design a divide and conquer algorithm to find the maximum and minimum of an array A of n elements, and prove that the algorithm makes at most $3n/2$ element to element comparisons.

Ans. Divide and conquer algorithm to find the maximum and minimum of an array A of n elements.

Suppose we know the maximum and minimum element in both of the $n/2$ sized partitions of an n-element ($n \geq 2$) array. Then in order to find the maximum and minimum element of the entire array we simply need to see which of the two maximum elements is the larger, and which of the two minimum is the smaller. We assume that in a 1-element array the only one element is both the maximum and the minimum element. Accordingly, we present the following algorithm for the maximum and minimum problem.

We assume the array, A is declared globally. The algorithm is invoked with low = 1 and high = n – that is, index of first element and index of last element respectively. And other two parameters are max and min which are initialized with first element.

MaxMin (low, high, max, min)

```
If (low = high), then    // case when n = 1
    max = min = A[low];
Else if (low = high – 1), then  // case when n = 2
    If (A[low] > a[high]), then
        max = A[low], min = A[high]'
    Else max = A[high], min = A[low];
    Endif
  Else // case when n > 2
    mid = (low + high)/2;
    MaxMin (low, mid, maxleft, minleft);
    MaxMin (mid + 1, high, maxright, minright);
    If (maxleft > maxright), then max = maxleft;
    Else max = maxright;
    If (minleft < minright), then min = minleft;
    Else min = minright;
```

16. Endif
17. Endif

Let $T(n)$ be the number of comparisons performed by the MaxMin algorithm. When $n = 1$, there are no comparisons – that is $T(1) = 0$. When $n = 2$, there is one comparison – that is, $T(2) = 1$. When $n > 2$, it is given as :

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$

MaxMin performs two recursive calls – one one p partition and another on r partition of the array, and then makes two further comparisons (steps 12 to 15) to arrange the maximum and minimum for the entire array.

Since we expect $T(n)$ to be a linear function, let $T(n) = an + b$.

We have $T(2) = 1$ and $T(4) = 2T(4/2) + 2 = 2T(2) + 2 = 2(1) + 2 = 4$.

Hence $2a + b = 1$ and $4a + b = 4$, from which we get $a = 3/2$ and $b = -2$.

$$\text{So } T(n) = \frac{3n}{2} - 2.$$

Q. 5. What is a stable sorting method ? Is mergesort a stable sorting method ?

Ans. Stable Sorting : Elements with the same value appear in the output array in the same order as they do in the input array. That is, ties between two elements are broken by the rule that whichever number appears first in the input array appears first in the output array. In other words, a sorting method is called stable if it preserves the relative order of equal keys in the array.

Example : Unsorted array : 22, 44, 77, 11, 44_2, 55, 66, 33

Sorted array : 11 22, 33, 44_1, 44_2, 55, 66, 77,

In this example, we have two equal keys (44). The relative positions of the keys 44_1 and 44_2 are retained in the sorted array.

Mergesort is stable. The elements comparison in merge includes equality so that the merge will be stable. Following figure illustrates the mergesort. Notice the elements : 44_1 and 44_2.

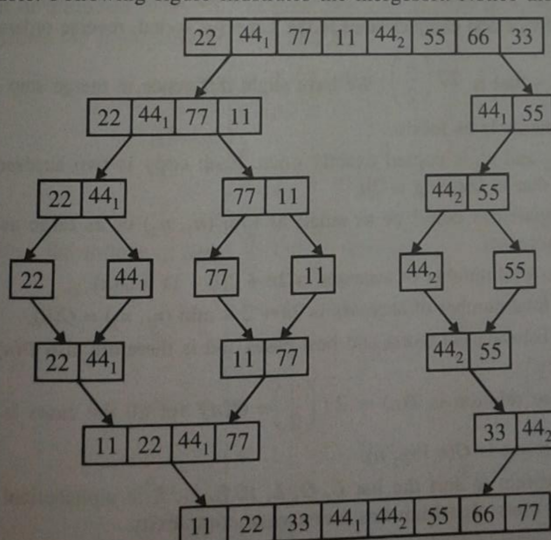


Fig. 4.10.

Q. 6. Determine the running time of mergesort for

(*i*) Sorted input

(*ii*) Reverse ordered input

(*iii*) Random ordered input.

Ans. Mergesort sorts a given array into increasing order as follows :

- Divide the array of size n, into two parts. For example

$$n_1 = \text{the first } n/2 \text{ elements in the array.}$$
$$n_2 = \text{the remaining elements in the array.}$$

- Sort n1 and n2 by recursively calling Mergesort with each one.
- Now we have two sorted arrays containing all the elements from the original array.

  Use merge procedure to combine them, put the result in the array.

  The best case running time of mergesort is with sorted input array. The worst case is with either reverse ordered input or random ordered input. The average case is between best and worst cases. The running time of the mergesort is given by the following recurrence relation :

$$T(1) = O(1) \qquad \text{for } n = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \qquad \text{for } n > 1$$

- **Divide :** The divide step just computes the middle of the subarray, which takes constant time, $O(1)$.
- **Conquer :** We recursively solve two subproblems, each of size $n/2$, which contributes $T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)$ to the running time.
- **Combine :** The merge procedure on an n-element subarray takes time $O(n)$.

  The time taken for divide and conquer steps is the same for sorted, reverse ordered, and random ordered input arrays – that is $2T\left(\frac{n}{2}\right)$. We have slight difference in merge step – that is, $O(n)$.

  The complexity of merge is as follows :

- Every element in n_1 and n_2 is copied exactly once. Each copy is two accesses, so the total number of accesses due to copying is $2n$.
- The number of comparisons could be as small as min (n_1, n_2) or as large as $(n - 1)$. Each comparison is two accesses.
- In the worst case, the total number of accesses is $2n + 2(n - 1) = O(n)$.
- In the best case, the total number of accesses is $2n + 2 \times \min(n_1, n_2) = O(n)$.
- The average case is between the worst and best cases and is therefore also $O(n)$.

  Hence, the recurrence relation is $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ for all the cases is the same. The running time of mergesort is $O(n \log_2 n)$.

Q. 7. Trace the Quicksort algorithm to sort the list C, O, L, L, E, G, E in alphabetical order. Give an instance, where the Quicksort algorithm has worst case complexity.

Ans. Trace of the Quicksort : Starting pivot = $(1 + 7)/2 = 4$

1	2	3	4	5	6	7	pivot	swap
C	*O*	*L*	*L*	*E*	*G*	*E*	*L*	*O, E*
C	*E*	*L*	*L*	*E*	*G*	*O*	*L*	*L, G*
C	*E*	*G*	*L*	*E*	*L*	*O*	*L*	*L, E*
C	*E*	*G*	*E*	*L*	*L*	*O*	*G*	*G, E*
C	*E*	*E*	*G*	*L*	*L*	*O*	Sorted	

The worst case occurs if given array arr[1...n] is already sorted. The parttitioning routine produces one subproblem with $n-1$ elements and one with 0 elements. The successive calls to partition will split array of length n, $n-1$, $n-2$, ..., 2 and running time proportional to $n + (n-1) + (n-2) + ... + 2 = [(n+2)(n-1)]/2 = O(n^2)$. The worst case also occurs if arr [1....n] starts out in reverse order.

Q. 8. List some of the relative advantages and disadvantages of the partition algorithm.

Ans. Relative advantages and disadvantages of the partition algorithm :

The best case behaviour of the quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length. In order to sort n elements, in this case the running time is in $O(n \log_2 n)$. This is because the recursion depth is $\log_2(n)$ and on each level there are n elements to be treated.

The worst case occurs when in each recursion step an ubbalanced partitioning is produced, namely that one part consists of only one element and the other part consists of the rest of the elements. Then the recursion depth is $n-1$ and quicksort runs in time $O(n^2)$. Both the best case and the average case are the same that is, $O(n \log_2 n)$.

The choice of the comparison element determines which partition is achieved. Suppose that the first element of the sequence is chosen as comparison element. This would lead to the worst case behaviour of the algorithm when the sequence is initially sorted. Therefore, it is better to choose the element in the middle of the sequence as comparison element.

Even better would it be to take the $\frac{n}{2}$th greatest element of the sequence (the median). Then the optimal partition is achieved. Actually, it is possible to compute the median in linear time.This variant of quicksort would run in time $O(n \log_2 n)$ even in the worst case.

Q. 9. To perform the multiplication of A and B

$$AB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

Ans. We define the following eight $\frac{n}{2}$ by $\frac{n}{2}$ matrices :

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} \qquad A_{12} = \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \qquad B_{12} = \begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} \qquad A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix} \qquad B_{22} = \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

The correctness of the above equations is easily verified by substitution.

$$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$$

$$P_2 = (A_{21} + A_{22}) \times B_{11} = \begin{bmatrix} 14 & 23 \\ 14 & 23 \end{bmatrix}$$

$$P_3 = A_{11} \times (B_{12} - B_{22}) = \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) = \begin{bmatrix} -3 & 2 \\ 5 & -20 \end{bmatrix}$$

$$P_5 = (A_{11} + A_{12}) \times B_{22} = \begin{bmatrix} 34 & 18 \\ 45 & 9 \end{bmatrix}$$

$$P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) = \begin{bmatrix} 3 & 27 \\ -18 & -18 \end{bmatrix}$$

$$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = \begin{bmatrix} 18 & 16 \\ 3 & 0 \end{bmatrix}$$

$$C_{11} = P_1 + P_4 - P_5 + P_7 = \begin{bmatrix} 17 & 22 \\ 21 & 18 \end{bmatrix}$$

$$C_{12} = P_3 + P_5 = \begin{bmatrix} 31 & 30 \\ 33 & 33 \end{bmatrix}$$

$$C_{21} = P_2 + P_4 = \begin{bmatrix} 11 & 25 \\ 19 & 3 \end{bmatrix}$$

$$C_{22} = P_1 + P_3 - P_2 + P_6 = \begin{bmatrix} 22 & 38 \\ 14 & 30 \end{bmatrix}$$

$$C = \begin{bmatrix} 17 & 22 & 31 & 30 \\ 21 & 18 & 33 & 33 \\ 11 & 25 & 22 & 38 \\ 19 & 3 & 14 & 30 \end{bmatrix}$$

OBJECTIVE TYPE QUESTIONS

1. What are the three sequential steps of divide-and-conquer algorithms ?
   (*a*) Combine conquer divide
   (*b*) Divide combine conquer
   (*c*) Divide conquer combine
   (*d*) Conquer divide conquer

2. The divide and conquer algorithm is closely tied to a type of recurrence relation. What is the base-case complexity of divide and conquer algorithm ?
   (a) $O(n)$ (b) $O(\log n)$
   (c) $O(n \log n)$ (d) $O(1)$
3. The divide and conquer algorithm will have ________ complexity.
   (a) $O(n)$ (b) $O(\log n)$
   (c) $O(n \log n)$ (d) $O(n^2)$
4. The divide and conquer strategy is used for problems such as sorting to reduce the complexity from ________.
   (a) $O(n)$ (b) $O(\log n)$
   (c) $O(n \log n)$ (d) $O(n^2)$
5. The average successful search time taken by binary search on a sorted array of 10 elements is
   (a) 2.6 (b) 2.7
   (c) 2.8 (d) 2.9
6. Merge sort performs in two phases : sort and merge. What is the expected time for merging ?
   (a) $O(n)$ (b) $O(\log n)$
   (c) $O(n \log n)$ (d) $O(n^2)$
7. Which one of the following is useful in implementing quicksort ?
   (a) Stack (b) Set
   (c) List (d) Queue
8. The recurrence relation that arises in relation with the complexity of binary search is (where k is a constant)
   (a) $T(n) = T(n/2) + k$ (b) $T(n) = 2T(n/2) + k$
   (c) $T(n) = T(n/2) + \log(n)$ (d) $T(n) = T(n/2) + n$
9. Which one of the following algorithm design techniques is used in Strassen's matrix multiplication algorithm ?
   (a) Dynamic programming (b) Backtracking approach
   (c) Divide and conquer strategy (d) Greedy method
10. In Strassen's matrix multiplication algorithm (C = AB), if the matrices A and B are not of type $2^n \times 2^n$, the missing rows and columns are filled with ________.
   (a) 0's (b) 1's
   (c) –1's (d) 2's
11. In Strassen's matrix multiplication algorithm ($C = AB$), the matrices A and B are decomposed into ________ blocks.
   (a) 2 (b) 4
   (c) 8 (d) 16
12. The median of n elements can be found in $O(n)$ time. Which one of the following is correct about the complexity of quick sort, in which median is selected as pivot ?
   (a) $Q(n)$ (b) $Q(n \log_2 n)$
   (c) $Q(n^2)$ (d) $Q(n^3)$

ANSWERS

1. (c)	2. (d)	3. (c)	4. (d)	5. (d)
6. (b)	7. (a)	8. (a)	9. (c)	10. (a)
11. (b)	12. (b)			

Chapter 5

Medians and Order Statistics

5.1. INTRODUCTION

INSIDE THIS CHAPTER

Order Statistics : i^{th} order statistics of a set of n elements is ith smallest element.

Medians : It is the middle element in the set of n elements such that if n is odd median is $\left(\frac{n+1}{2}\right)^{th}$ element, otherwise if n is even then middle element lies at $i = \frac{n}{2}$ and $i = \frac{n}{2} + 1$.

Example 5.1. Write an algorithm MEDIAN (S) to get the median element from the sequence S of n elements.

Algorithm 5.1. Median (S)

```
n ← length [S]
if (n % 2 = 0) then
i ← ⌊(n + 1)/2⌋
j ← ⌈(n + 1)/2⌉
else
i ← ⌈(n + 1)/2⌉
```

Time complexity if $O(1)$, as middle element can be found in constant time.

Example 5.2. Find the median element in the following set S of elements :

1 9 3 3 4 5 6 7 7 8

Solution.

1	2	3	4	5	6	7	8	9	10
1	9	3	3	4	5	6	7	7	8

Here, n = 10, *i.e.*, even

So middle element lies at $\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{11}{2} \rfloor = 5$

and $\lceil \frac{n+1}{2} \rceil = \lceil \frac{11}{2} \rceil = 6$.

i.e., 5th and 6th elements are median

i.e., median is 4, 5

Example 5.3. Write an algorithm to find the minimum and max. of elements in an array. *A*. [1 .. *n*].

Algorithm 5.2. Minimum (*A*)

```
1. min ← A(1)
2. for i ← 2 to length [A]
3.     do if min > A[i]
4.         then min ← A[i]
5. return min

Maximum (A)
1. max ← A(1)
2. For i ← 2 to length [A]
3.          do if max < A[i]
4.                  then max ← A[i]
5. return max.
```

Running time of above algorithm is $\theta(n)$, as $n - 1$ comparisons are performed in worst case.

Example 5.4. Show that the expected running time of line 4, in above algorithm $\theta(\lg n)$.

Solution. Assume that all numbers in A are randomly drawn from the interval [0, 1]. Let S_1, S_2, ..., S_n be n random variables where S_i represents the no. of items (0 or 1) that line 4 is executed during the ith iteration of the loop. Where

$$S = S_1 + S_2 + S_3 + ... + S_n$$

We have
$$E[X] = \sum_{i=0}^{\infty} P_r\{X \geq i\}$$

Probability that maximum is found out of n elements each time is $\frac{1}{n}$, for each random variable S_i since MAX or MIN algo runs in $O(n)$ time, total expected time is

$$E(O(n_i)) = O([E(n_i)])$$

$$\sum_{i=0}^{n-1} O[E(n_i)] = O\left(\sum_{i=0}^{n-1} E(n_i)\right)$$

from above formula

$$= O\left(\sum_{i=0}^{n-1} P_r\{X \geq i\}\right) = \Theta\left[\sum_{i=0}^{n}\left(\frac{1}{n}\right)\right] = O(\log n)$$

Finding Maximum and Minimum

Algorithm 5.3. Max_min (A, n, max, min)

```
1. max ← min ← A[1]
2. for i ← 2 to n do
       if (A[i] > max) then max ← A [i]
       else if (A[i] < min) then min ← A[i]
```

Best case occurs when the elements are in the increasing order. The no. of element comparisons is $n - 1$. The worst case occurs when the element are in decreasing order. In this case the number of element comparisons is $2(n - 1.)$ The average number if element comparisons is less than $2(n - 1)$. on the average, $A[i]$ is greater than max half the time, and so the average number of comparisons is $\frac{3n}{2} - 1$.

Example 5.5. Show through a recursive MAXIMIN algo that $\lfloor \frac{3n}{2} \rfloor - 2$ comparisons are necessary in the worst case to find both the maximum and minimum of n numbers.

Solution.

Algorithm 5.4. MAXMIN2 (i, j, max, min)

```
1.     if (i = j) then max ← min ← A[i]
2. else if (i = j - 1) then
       if (A[i] < A[j]) then
       max ← A[j]
       min ← A[i]
   else
       max ← A[i]
       min ← A[j]
3. else mid = ⌊(i+j)/2⌋
       Maxmin2 (i, mid, max, min)
       Maxmin2 (mid + 1, j, max1, min1)
       if (max < max1) then max ← max1
       if (min > min1) then min ← min1
```

Let us simulate MaxMin2 on the following set of elements

	1	2	3	4	5	6	7	8	9
A :	22	13	–5	–8	15	60	17	31	47

Here, $i = 1, j = 9$, max = 60, min = –8

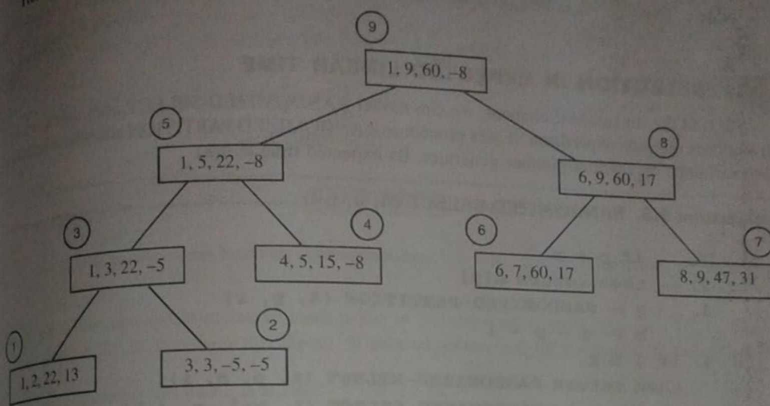


Fig. 5.1

The circled numbers in the upper left or right corner of each node represents the order in which max and min are assigned values.

Now the number of elements comparisons in MaxMin2 is given by recurrence relation $T(n)$,

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$, for some +ve integer then,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 2 \\ &= 2\left(2T\left(\frac{n}{4}\right) + 2\right) + 2 \\ &= 4T\left(\frac{n}{4}\right) + 4 + 2 \\ &\vdots \qquad \vdots \\ &= 2^{k-1}\, T(2) + \sum_{1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 2^k\left(\frac{1}{2} + 1\right) - 2 \\ &= \frac{3}{2} 2^k - 2 \\ &= \frac{3}{2} n - 2 \text{ as } 2^k = n \end{aligned}$$

Note : That $\frac{3n}{2} - 2$ is best, average and worst case number comparisons when n is a power of two.

5.2. SELECTION IN EXPECTED LINEAR TIME

To find the ith smallest element, we can model RANDOMIZED-SELECT after quicksort algo. It works on one side of partition. It uses procedure RANDOMIZED PARTITION and whose behaviour is determined by random-number generator. Its expected time is $\theta(n)$

Algorithm 5.5. RANDOMIZED-SELECT (A, p, r, I) ....

```
1.    if p = r
2.    then return A[p]
3.    q ← RANDOMIZED-PARTITION (A, p, r)
      k ← q - p + 1
4. if j ≤ k
   then return RANDOMIZED-SELECT (A, p, q, i)
5. else return RANDOMIZED-SELECT (A, q+1, r, i-k)
```

After RANDOMIZED Partition is executed, the array $A[i \,..\, r]$ is partitioned into two non-empty Subarray $A[p \,..\, q]$ and $A[q + 1 \,...\, r]$ such that each element of $A[p \,..\, q]$ is less than each element of $A[q + 1 \,..\, r]$. The algorithm computes the number k of elements in the subarray $A[p \,..\, q]$. The algorithm now determines in which of the following two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ the ith smallest element lies. If $i \le k$, then the desired element lies on the low side of the partition, and it is recursively selected from the subarray. If $i > k$ then it is on high side.

Algorithm RANDOMIZE-PARTITION produces a partition whose low side has 1 element with probability $\frac{2}{n}$ and i element with probability $\frac{1}{n}$ for $i = 2,3, \ldots n - 1$. If it is unlikely the ith element lies on larger side. Thus we have recurrence relation.

$$T(n) \le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n)$$

$$\le \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

$$= \frac{2}{n}\sum_{\lceil n/2 \rceil}^{n-1} T(k) + O(n) \qquad \left\{ \max(k, n-k) = \begin{cases} k & \text{if } k \ge \frac{n}{2} \\ n-k & \text{if } k < \frac{n}{2} \end{cases} \right.$$

by induction,

$$T(n) \le \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n)$$

$$\le \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil} k\right) + O(n)$$

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + O(n)$$

$$= c(n-1) - \frac{c}{n}\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right) + O(n)$$

$$= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n)$$

$$\leq cn, \quad \text{thus } O(n)$$

EXERCISES

1. Show that no algorithms based on comparisons uses less than $\frac{3n}{2} - 2$ comparisons i.e., MaxMin2 is optimal?
2. Show that how quicksort can be made to run in $O(nlgn)$ time in worst case.
3. Analyze SELECT to show that the no. of element greater that the median-of-medians x and the no. of elements less than x is at least $\lceil \frac{n}{4} \rceil$ if $n \geq 38$.
4. Let $X[1 .. n]$ and $Y[1 .. n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(nlgn)$ time algorithm to find median of all $2n$ elements in arrays X and Y.
5. Find the best solution for 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

❑❑❑

Chapter 9

Graphs

9.1. REPRESENTATION OF GRAPHS

INSIDE THIS CHAPTER

Before we give representation of graph let us provide the definition of adjacency matrix for a graph.

Adjacency Matrix

If $G(V, E)$ is a graph then adjacency matrix is $|V| \times |V|$ matrix, *i.e.* vertex to vertex such that matrix $A = [a_{ij}]$ and

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

i.e., if there is an edge from V_i to V_j then entry to a_{ij} is 1, otherwise O. See Fig. 9.1.

Its matrix is given by

$$\begin{array}{cc} & \xrightarrow{\quad V \quad} \\ & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ V\downarrow \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

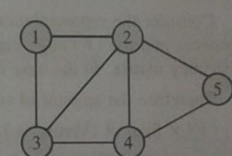


Fig. 9.1

Adjacences List

Linked list representation of the given graph in Fig. 9.1, having above adjacency matrix is given by

9.2.1. Difference between BFS and DFS

BFS	DFS
1. The starting vertex is given to traverse a graph.	1. No starting vertex is given.
2. Uses queues to traverse a graph.	2. Uses stacks to traverse a graph.
3. A parent vertex is traversed first compared to its child vertex.	3. A child vertex is traversed first compared to its parent vertex.
4. It is used to find spanning tree as well as shortest path.	4. It is used to sort a graph.
5. BFS doesn't uses recursion or backtracking.	5. DFS uses backtracking.

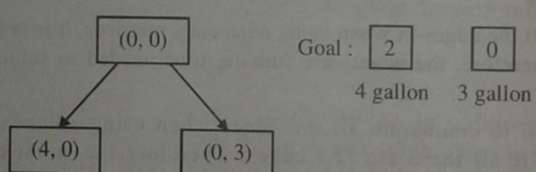
9.2.2. Breadth First Search

As the name implies, before proceeding to the next level of tree or forest, nodes at current level are expanded first. This process is continued till the goal state is reached.

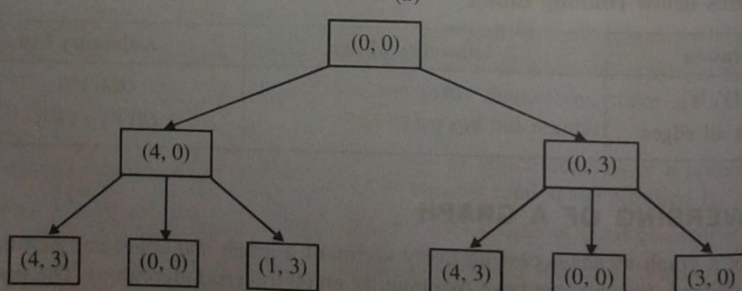
9.2.2.1. Water-JUG Problem

Let us consider that there are two Jug one of which can store 4-Gallons of water and the other 3-Gallons of water we have to find a solution state such that the 4-Gallons Jug contains 2-Gallon, and 3-Gallons Jug contains no water (2, 0). We can throw the water and there is no scale for measurement on the jug.

We can find the solution to above problem using BFS in terms of tree. Thus, to obtain the solution state (2, 0), we have to expand all the nodes upto level 6, such that it forms a complete tree. See Fig. 9.3 below

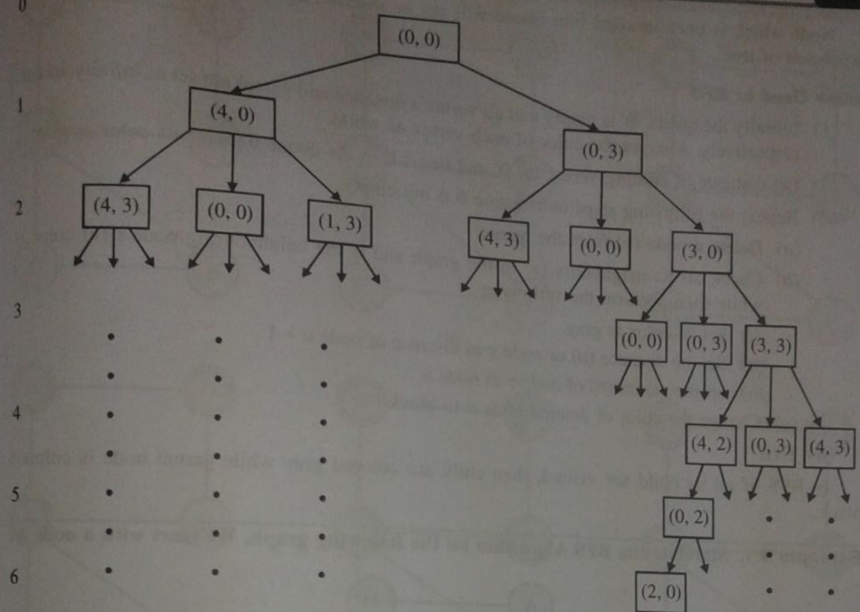


(*a*)



(*b*)

Fig. 9.3



(c)

Fig. 9.3.

Figure 9.3 shows breadth first expansion of water jug problem.

9.2.2.2. BFS Procedure

Thus BFS expands the discovered or undiscovered node uniformly across the breadth. In order to mark the discovered and undiscovered node, colors are given as white, gray and black.

(1) Node which is not visited is white.

(2) Node which is visited but its child node are not visited is gray.

(3) If all of its child nodes are visited than make it black.

In BFS, we have to store the following for every vertex in the graph.

(*a*) Color of the vertex as color [*V*].

(*b*) Distance of the vertex from starting vertex *i.e.* $d[V]$.

(*c*) Parent of the vertex *i.e.* $\pi[V]$.

Let us consider the following undirected forest on which BFS method is applied using coloring techniques just described. Let us assign 0 value to the start node and infinity to all other nodes. Then measure the length from root to other levels as 1, 2, 3, ... let also manage a queue, which consists of first element as node which is to be currently visited.

Example 9.2. Simulate the Algorithm on the following graph with S as a start node.

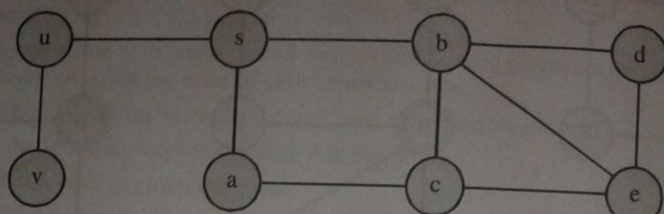


Fig. 9.7.

Table 9.2. Trace of BFS.

#	Node Deleted	Queue	White Nodes	Grey Node	Block Nodes
Initial	—	*S*	*u, v, a, b, c, d, e*	*S*	—
1	*S*	*u, a, b*	*v, c, d, e*	*u, a, b*	*S*
2	*u*	*a, b, v*	*c, d, e*	*a, b, v*	*S, u*
3	*a*	*b, v, c*	*d, e*	*b, v, c*	*S, u, a*
4	*b*	*v, c, d, e*	—	*v, c, d, e*	*s, u, a, b*
5	*v*	*c, d, e*	—	*c, d, e*	*s, u, a, b, v*
6	*c*	*d, e*	—	*d, e*	*s, u, a, b, v, c*
7	*d*	*e*	—	*e*	*s, u, a, b, v, c, d*
8	*e*	—	—	—	*s, u, a, b, v, c, d, e*

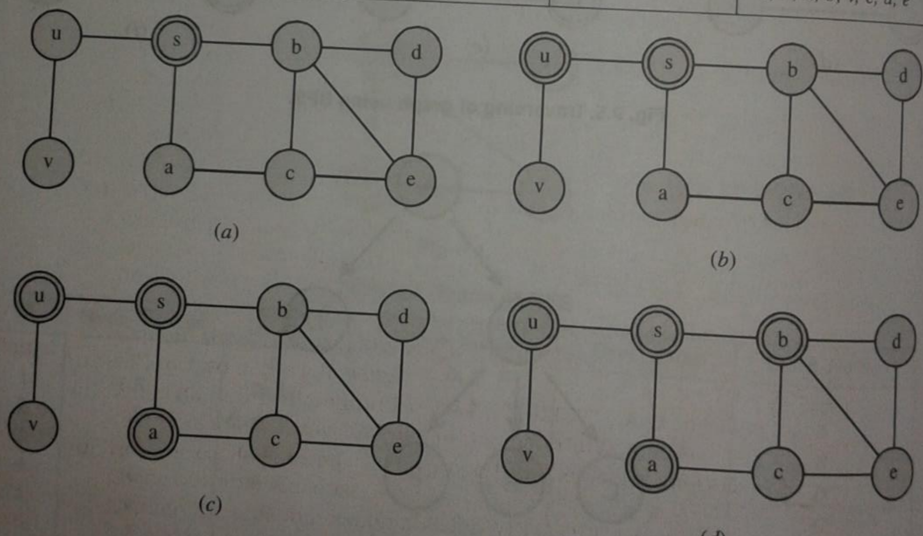


Fig. 9.8.

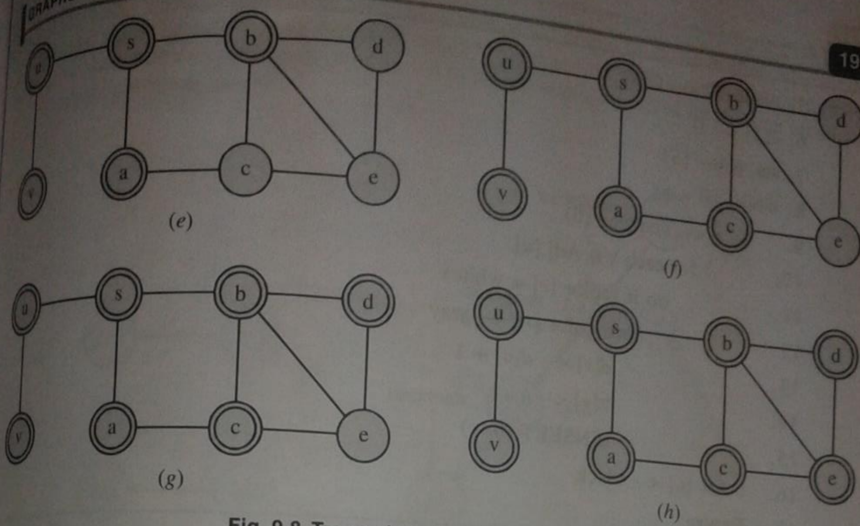


Fig. 9.8. Traversing of graph using BFS.

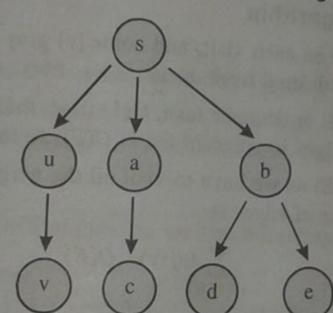


Fig. 9.9. BFS spanning tree.

NOTE

To draw the BFS tree we have to check that wheneven we deleted any vertex (a) from the queue during tracing of BFS algorithm, how many and which are the vertices (b) are inserted in the queue, then there is an edge a → b exists in BFS tree.

9.2.2.3. BFS Algorithm

Algorithm 9.1. BFS (G, S)

```
For each vertex u ∈ v(G) – S
    do color [u] <-- white
        d[u] <-- ∞
        π[u] <-- Nil
```

```
color (S) <-- Gray
d[S] <-- 0
set θ <-- {S}
while (θ ≠ φ)
    do u <-- dequeue (θ)
        for each v ∈ Adj [u]
            do if (color [v] = white)
                then color [v] <-- gray
                    d[v] <-- d[u] + 1
                    π[v] <-- u
                    θINSERT (θ, v)
    color [u] <-- black
Return
```

9.2.2.4. Analysis of BFS Algorithm

We increment the path length on each visit, and color $[v]$ gray and this child node is inserted into queue, and the black node is deleted from queue.

Each θINSERT and θDELETE or dequeue take, $\Theta(1)$ time, thus total time for queue operation is $O(v)$. Time taken to scan the edges in adjacent list is $O(E)$, so total time taken is $O(v + E)$.

Time to scan the edges is $O(E)$ as we have to visit all the neighbouring node of all the vertex and which is equivalent to number of edges is

$$\sum_{v \in V(G)} Adj(v) = O(E)$$

9.2.3. Depth First Search

In depth first search method, we move to depth of one branch before exploring the other childrens at the same level. If one branch is fully explored then we backtrack and explore other branch to its depth. This process is continued till all the branches are explored to their depth in a forest.

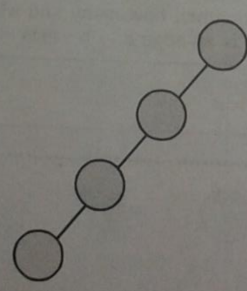


Fig. 9.10. Depth first exploration of a node.

The input of DFS algorithm can be directed or undirected graph. In the algorithm, we maintain start time and finish time of a node called timestamp, *i.e.* when we visit a node for first time, we mention the start time when this node is explored to its full depth, *i.e.* all of its children are visited then we mentioned finish time.

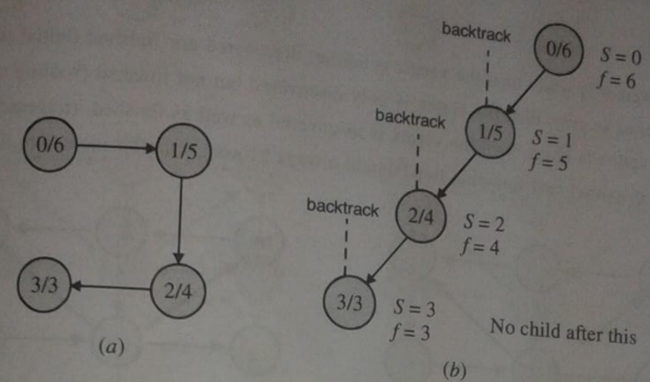


Fig. 9.11. DFS exploration showing start/finish time.

We assume that, it takes unit time to visit a node. If we want, we can use the coloring techniques along with this to keep track of visisted and explored node. Thus we can write DFS algorithm as follows.

Example 9.3. Simulate the DFS algorithm on the following graph.

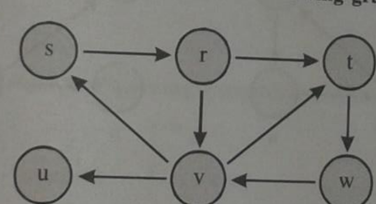


Fig. 9.12.

Solution. Assume that the starting vertex is S.

Steps Used in DFS :

1. Initially all the vertices of graph have its color white.
2. Assume a starting vertex and assign a discovery time (d) to this vertex and change its color to grey.
3. Repeat until all the vertices of graph is not traversed.
   (*a*) Check the neighbours of the vertex, which is just discovered (say *V*).

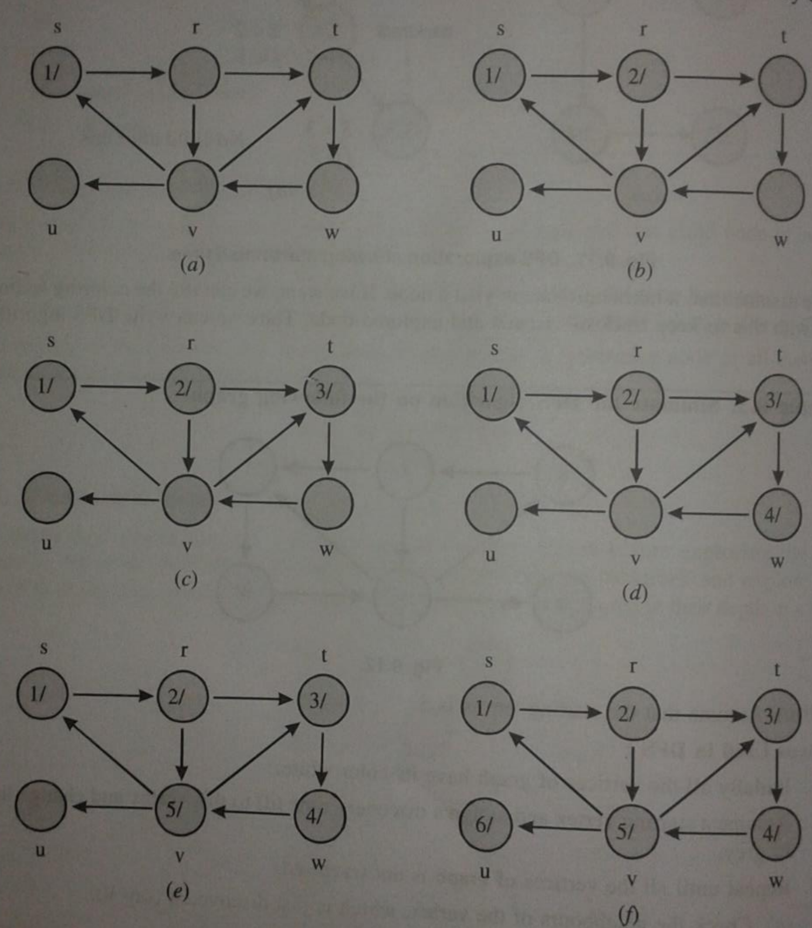
(*b*) If the neighbouring vertex of 'V' is not discovered then discover it by assigning a discovery time to it and change its color to grey.

(*c*) If all its neighbours of vertex 'V' are already discovered then finish the vertex by assign a finishing time and change its color to black.

4. Exit.

Note :

1. If a vertex is white then the vertex is neither discovered nor finished (initial state).
2. If vertex is grey, then the vertex is only discovered but not finished (waiting state).
3. If a vertex is black, then the vertex is discovered as well as finished. (travesed state).
4. The discovery and finishing time should always be assigned after increment it by one.



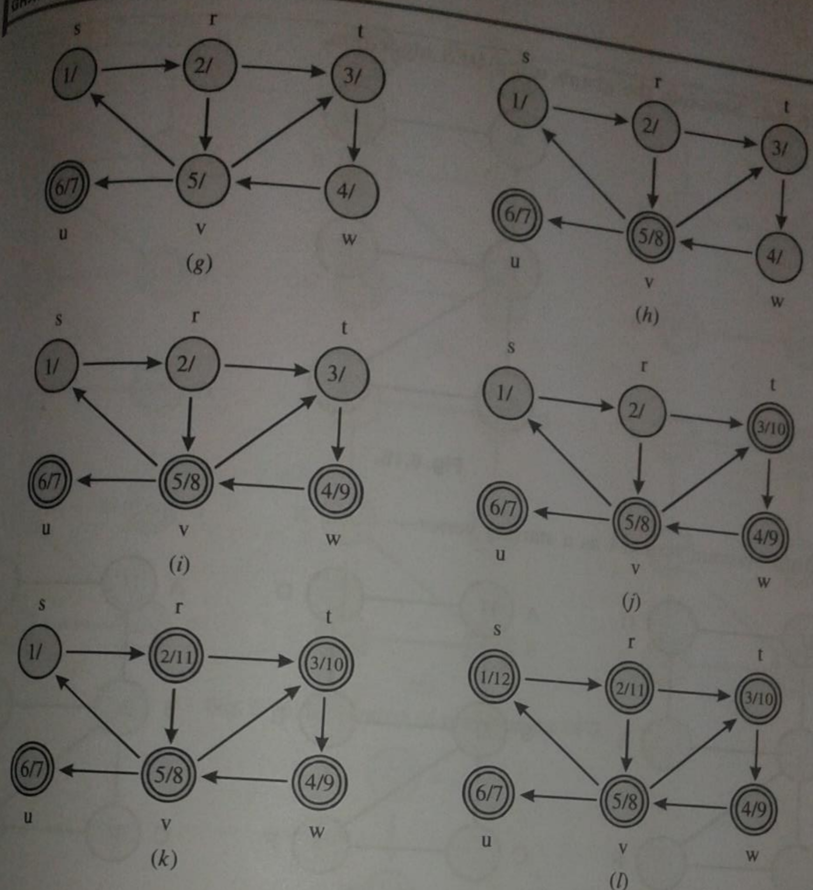


Fig. 9.13. Traversing of graph using DFS.

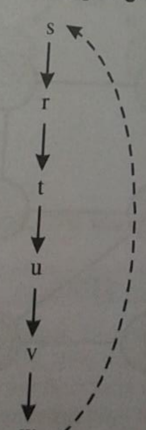


Fig. 9.14. DFS tree/spanning tree.

Example 9.4. Simulate the graph using DFS algorithm.

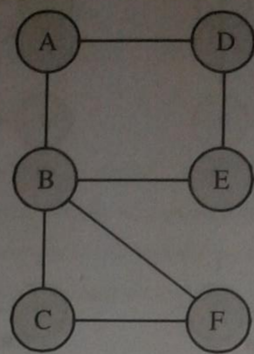
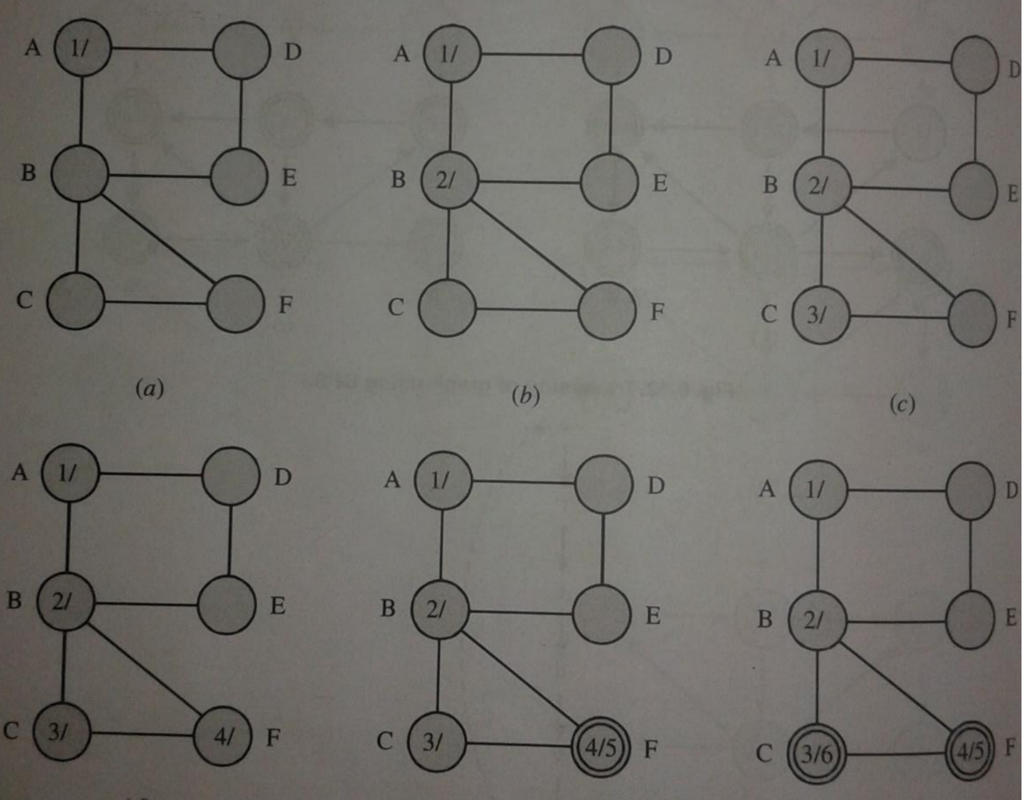


Fig. 9.15.

Solution. Assume vertex A as a starting vertex



(*a*) (*b*) (*c*) (*d*) (*e*) (*f*)

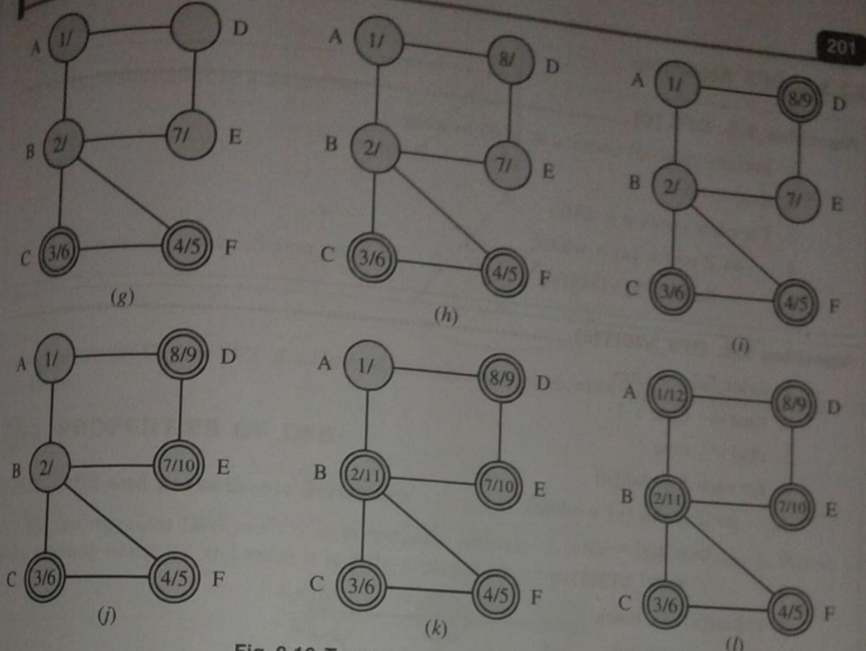


Fig. 9.16. Traversing of graph using DFS.

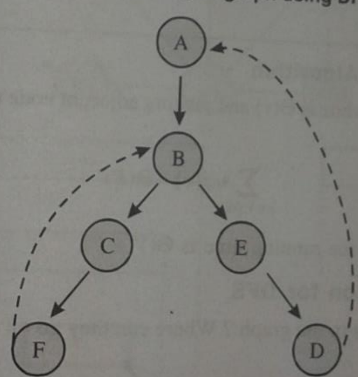


Fig. 9.17. DFS spanning tree.

NOTE

The most natural result of a DFS of graph is a spanning tree of the vertices reached during the search Fig. 9.17 shows the DFS spanning tree for the undirected graph (Fig. 9.15). The root of the trees is A, the first node visited. In fact, we can select any one of the node as a root. Dashed lines represent back edges. The back edges in a graph are those that connect some descendent node in a tree to an ancestor node in the same tree. It can be shown that if the graph is undirected then all of its edges are tree edges or back edges.

9.2.3.1. DFS Algorithm

Algorithm 9.2. DFS (*G*)

```
Initially color all vertex u ∈ V(G) to white
time <-- 0
For each vertex u ∈ U(G)
    do if (color [u] = white)
        then DFS_VISIT(u)
```

Algorithm 9.3. DFS_VISIT(u)

```
color [u] <-- Gray
time <-- time + 1
d[u] <-- time
for each V ∈ Adj[u]
    do if (color [v] = white)
        then π[v] <-- u
            DFS_VISIT(v)
colour[u] <-- black
time <-- time + 1
f[u] <-- time
```

9.2.3.2. Analysis of DFS Algorithm

Running time for coloring white is $\Theta(v)$ and visiting adjacent node means adjacent edges requires $\Theta(E)$ time as

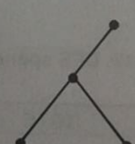
$$\sum_{v \in V(G)} Adj[v] = \Theta(E)$$

and since both are required so the running time is $\Theta(V + E)$.

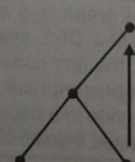
9.2.3.3. Edge Classification for DFS

What about the other edges in the graph ? Where can they go on a search ? Every edge is either

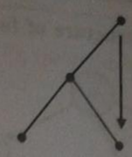
1. A tree edge



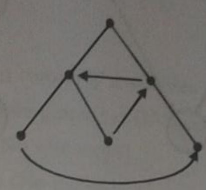
2. A back edge to an anchestor



3. A forward edge to a decendent



4. A cross edge to a different node



on any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

9.3. PROPERTIES OF DFS

9.3.1. DFS and parenthesis Structure

We can represent DFS method as parenthesis structure in which when a vertex is visited we write opening brace '(' and when it is fully explored we write closing brace ')'.

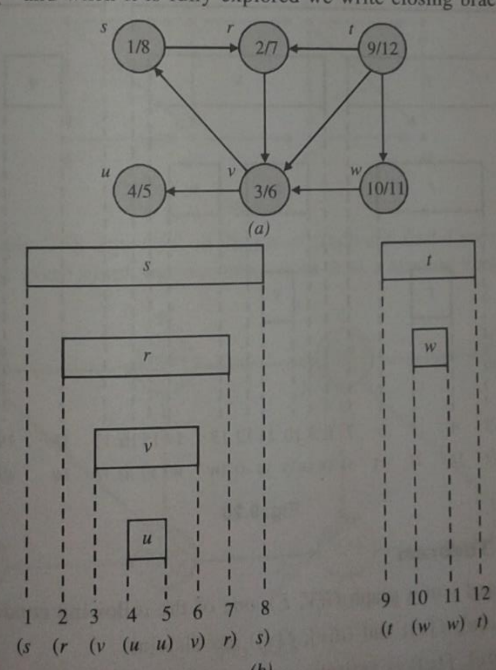


Fig. 9.18. (a) Shows results of DFS. (b) Shows the order in which tree nodes are expanded with their opening and closing brace according to time.

Example 9.5. Show the parenthesis structure of following graph

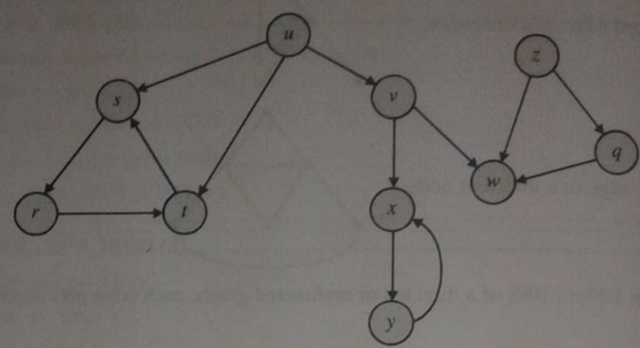


Fig. 9.19.

Solution.

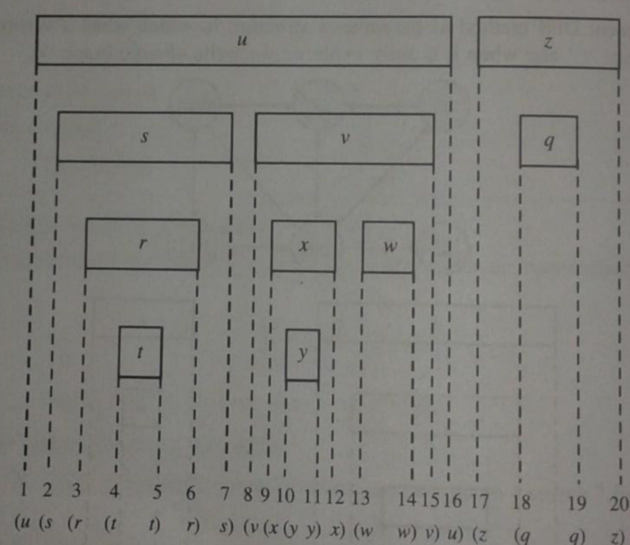


Fig. 9.20

9.3.2. Parenthesis Theorem

For two vertices u and v of a graph $G(V, E)$, one of the following conditions may holds :

(*a*) The intervals $(d[u], f[u])$ and $(d[v], f[v])$ are disjoint.

(*b*) The interval $(d[u], f[u])$ is entirely within $(d[v], f[v])$.

(*c*) The interval $(d[v], f[v])$ is entirely within $(d[u], f[u])$.

9.3.3. White Path Theorem

For two vertices u and v of a graph $G(V, E)$, the vertices in the path from u to v are white, when vertex u is discovered.

9.4. APPLICATIONS OF DFS

9.4.1. Topological Sorting

A graph $G(V, E)$ is said to be topological sorted, if it satisfies the following conditions :

(*i*) The graph G should be directed acyclic graph *i.e.* DAG.

(*ii*) The graph G should be irreflexive (No self loops).

(*iii*) The graph G should be assymmetric (No cycles).

(*iv*) The graph G should be transitive.

A graph $G(V, E)$ is topological sorted, if all its vertices are arranged in a horizontal line and all the edges used to join two vertices in the sorted graph, should always have its direction from left to right.

Example 9.6. Given a graph G, with $|V|$ number of vertices and $|E|$ number of edges.

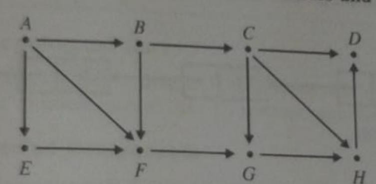


Fig. 9.21

Solution. Step I : Apply DFS algorithm on the given graph and find discovery and finishing time of all the vertices in the given graph by selecting vertex A as a starting vertex.

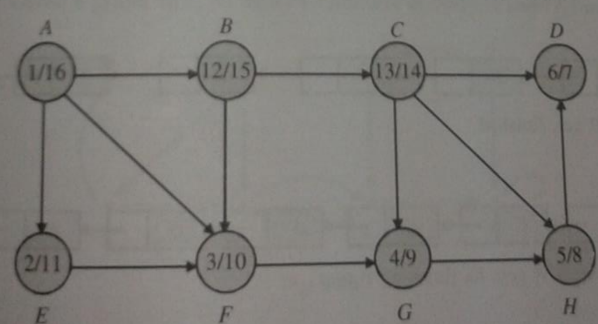
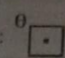
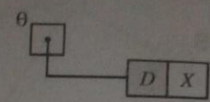


Fig. 9.22

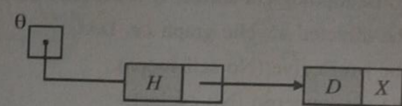
Step II : Create a queue θ using a linked list with front and rear pointers and as the vertex in step-I gets its finishing time, insert it onto the front of a queue, θ.

(*a*) Initially ueue θ is empty : 

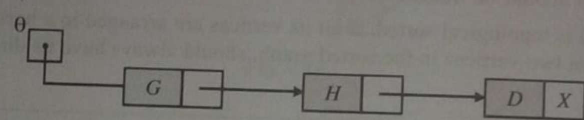
(*b*) Vertex *D* gets its finishing time, insert it onto the front as



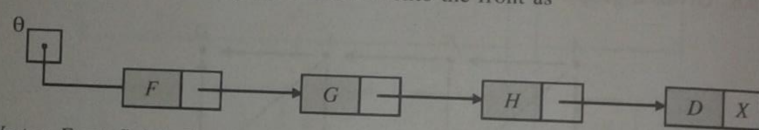
(*c*) Vertex *H* gets finished.



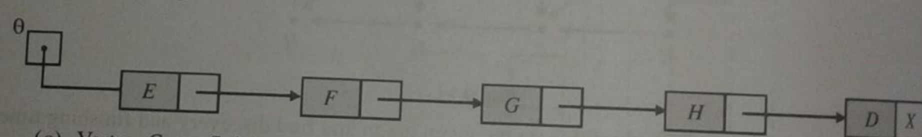
(*d*) Vertex *G* got finished.



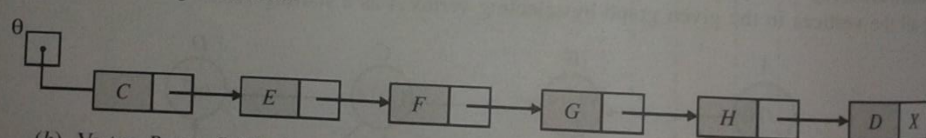
(*e*) Vertex *F* gets its finishing time. Insert it onto the front as



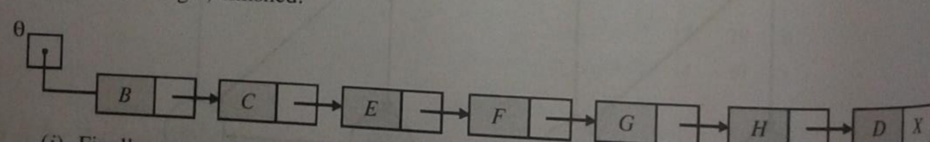
(*f*) Vertex *E* get finished.



(*g*) Vertex *C* got finished.



(*h*) Vertex *B* get, finished.



(*i*) Finally vertex *A* gets its finishing time.

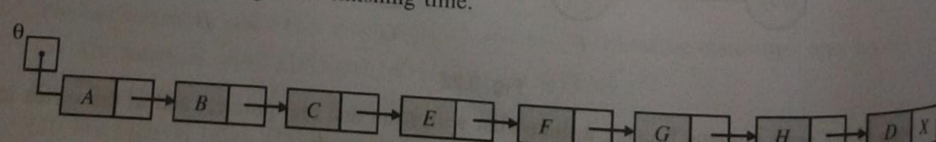


Fig. 9.23

Step III : Return the created queue, θ and attached all the edges between all the vertices as given in the original graph.

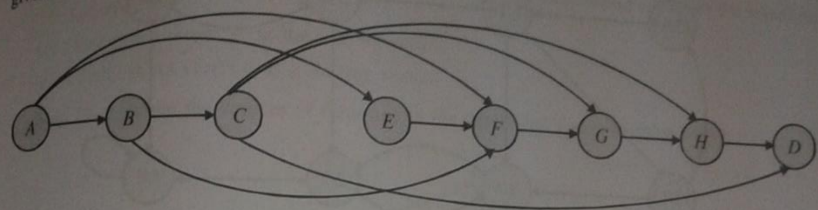


Fig. 9.24. Topological Sorted Graph

9.4.1.1. Algorithm of Topological Sorting

Algorithm 9.4. Topological-Sort (*G*) ....

(1) CALL DFS (*G*) to compute finishing time $f[v]$ for each vertex v.

(2) As each vertex is finished insert it onto the front of a linked list based queue.

(3) Return the linked list of vertices.

9.4.1.2. Analysis of Topological Sort

We can perform a topological sort in time $\Theta(V + E)$, since depth-first. Search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

9.4.2. Strongly Connected Components

A strongly connected component (SCC) of a directed graph $G(V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in c, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is vertices u and v are reachable from each other.

Example 9.7. Given a graph $G(V, E)$ with $|V|$ number of vertices and $|E|$ number of edges

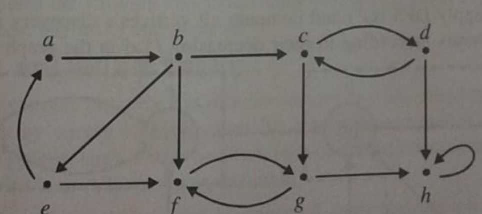


Fig. 9.25

Solution.

Step I : Apply DFS(G) on the given graph $G(V, E)$ to compute the discovery and finishing time of each vertex in the graph. Assume vertex '⊂' as a starting vertex.

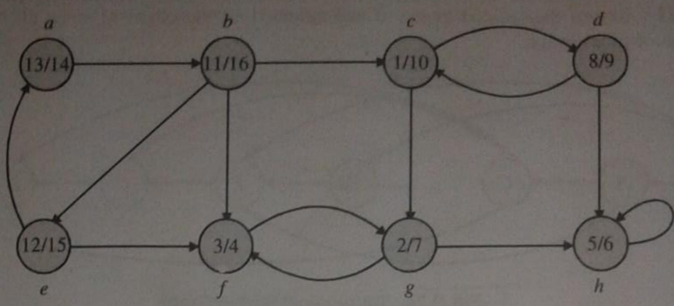


Fig. 9.26

Step II : Transpose the given graph G (V, E) and thus generates a new graph G^T such that

$$G^T = (V, E^T), \text{ where}$$

$$E^T = \{(u, v) : (v, u) \in E\}$$

i.e. E^T consists of the edges of G with their directions reversed.

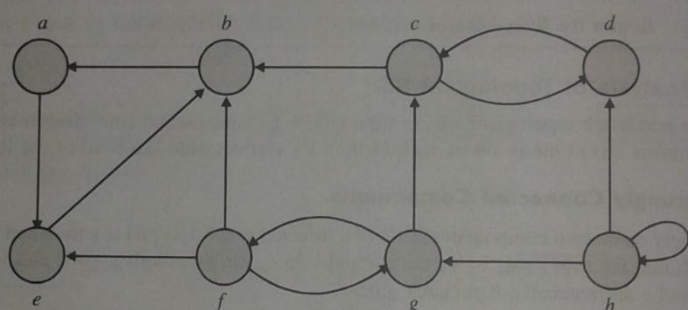


Fig. 9.27

Step III : Again apply DFS (G^T) and compute all vertices's discovery and finishing time by selecting the starting vertex, according to their decreasing $f[u]$ in the graph G.

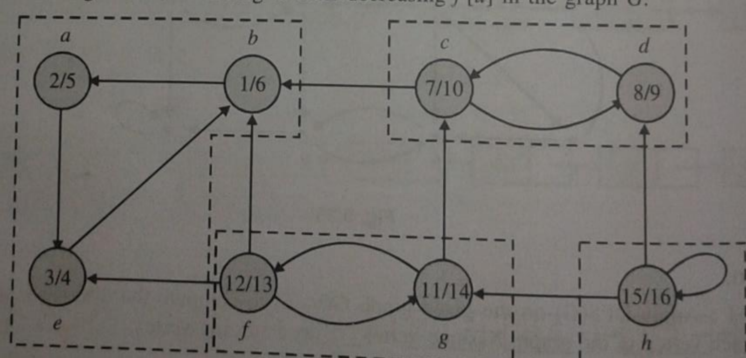


Fig. 9.28

(a) Select vertex '*b*' as a starting vertex, as its $f[v]$ is maximum in graph *G*.

(b) Now select vertec '*c*' as a starting vertex as it have a maximum $f[v]$ in all the remaining vertices.

(c) Now select vertex '*g*' as the starting vertex.

(d) Finally select vertex *h* as a starting vertex.

Step IV : Output the vertices of each tree in the depth first forest as show below :

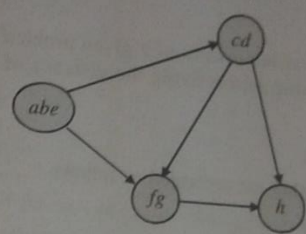


Fig. 9.29. Strongly-connected components.

9.4.2.1. Algorithm for Strongly Connected Components

Algorithm 9.6. Strongly Connected Components (*G*)

(1) Call DFS (G) to compute finishing times $f[u]$ for each vertex u.

(2) Transpose the graph GF and compute G^T.

(3) Call DFS (G^T), but in the main loop of DFS. Consider the starting vertex in order of decreasing $f[u]$.

(4) Output the vertices of each tree in depth-first forest formed in line 3 as a seperate SCC.

9.4.2.2. Analysis of Strongly Connected Components

Given an adjacency list representation of G, the time to create G^T is $O(V + E)$. It is interesting to observe that G and G^T have exactly the same SCC. The following linear-time (*i.e.* $\Theta(V + E)$ time) algorithm computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and other on G^T.

9.5. MINIMUM SPANNING TREES

In the design of electronic circuitry it is often necessary to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of $(n - 1)$ wires, each connecting two pins of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v. We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight is minimized.

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

Since T is acyclic and connects all of the vertices, it must form a tree or just a connected graph, which we can all a spanning tree. We can call the problem of determining T as the minimum-spanning tree problem *i.e.* a spanning tree with minimum cost or weight $w(T)$ called minimum spanning tree (MST).

In this session, we will examine two algorithm for solving the minimum spanning tree problem *i.e.*

(*a*) Kruskal's Algorithm

(*b*) Prim's Algorithm

9.5.1. Kruskal Algorithm

Consider the arcs of the distance network of a given problem. If the network is undirected, then it is sufficient to form a table summarizing the distance of each arc (i, j) in the network, only for $i < j$.

9.5.1.1. Kruskal Algorithm

The step of Kruskal's algorithm are presented as follows :

Step 1 : Input the following :

- Number of nodes in the network, n.
- Distance matrix (if there is no arc between any two nodes, then set the distance between them to infinity/a very high value. Also set the diagonal entries as infinity/a very high value.

Step 2 : Arrange the arc along with their distance as per ascending order of their distances let this arrangement be called as set M.

Step 3 : Create a graph with n nodes without arcs.

Step 4 : Select the next undeleted arc (first undeleted arc on first pass of this statement) from the set M.

Step 5 : Check whether the inclusion of the selected arc in the previous step into the graph forms a cycle. If yes, go to step 7, otherwise goto step 6.

Step 6 : Include the selected arc into the graph.

Step 7 : Delete that arc from the set M.

Step 8 : Check whether all the nodes in the graph are connected. If no, go to step 4, otherwise go to step 9.

Step 9 : Stop.

Example 9.8. Consider the networks given below. Find the minimum spanning tree using Kruskal's algorithm ?

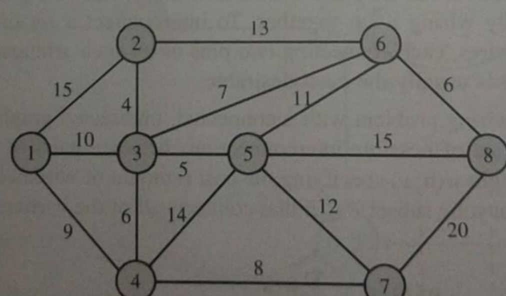


Fig. 9.30.

Solution.

Step 1 : Since the network is undirected, the distance of each arc (i, j), for $i < j$ is presented as in Table below

Arc	Distance	Arc	Distance
1-2	15	4-5	14
1-3	10	4-7	8
1-4	9	5-6	11
2-3	4	5-7	12
2-6	13	5-8	15
3-4	6	6-8	6
3-5	5	7-8	20
3-6	7		

Step 2 : Sort the distances of the arcs as per ascending order as shown below

Arc	Distance	Arc	Distance
2-3	4	5-6	11
3-5	5	5-7	12
3-4	6	2-6	13
6-8	6	4-5	14
3-6	7	1-2	15
4-7	8	5-8	15
1-4	9	7-8	20
1-3	10		

Step 3 : Figures below shows step 3 onwards. The arcs from 2-3 to 1-4 from the table above are induded one by one. The tree in last figure is the minimum spanning tree and the corresponding sum of the distances of the arcs in it is 45.

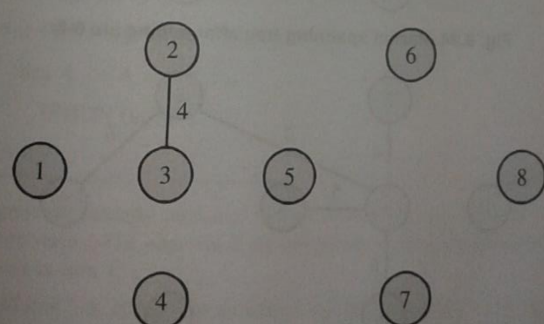


Fig. 9.31. Partial spanning tree after adding arc 2-3.

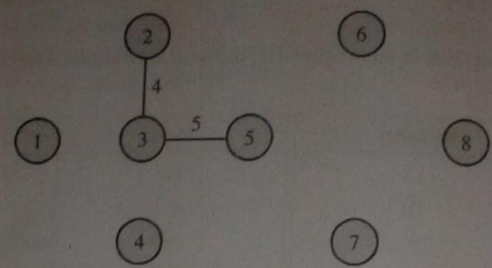


Fig. 9.32. Partial spanning tree after adding arc 3-5

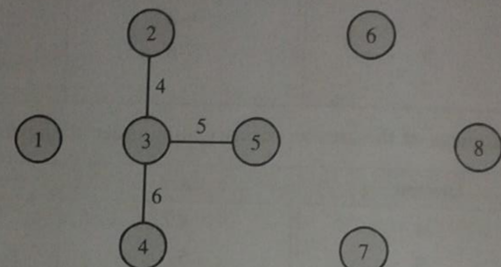


Fig. 9.33. Partial spanning tree after adding arc 3-4

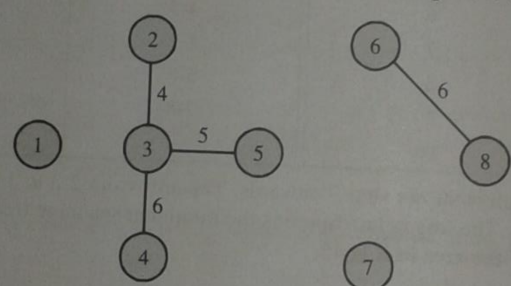


Fig. 9.34. Partial spanning tree after adding arc 6-8.

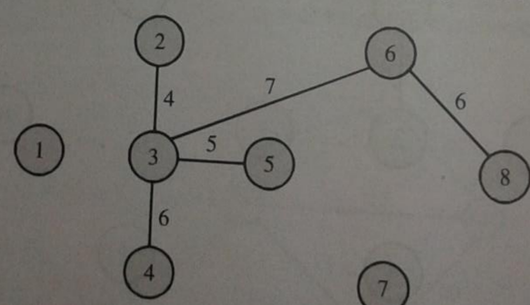


Fig. 9.35. Partial spanning tree after adding arc 3-6.

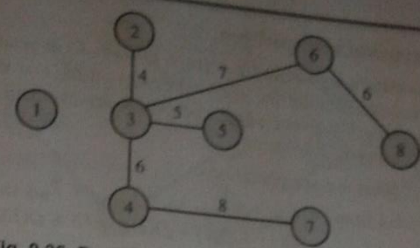


Fig. 9.36. Partial spanning tree after adding arc 4-7.

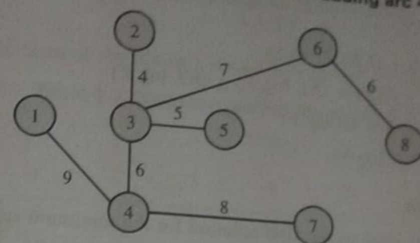


Fig. 9.37. Complete spanning tree after adding arc 1-4.

9.5.1.2. Kruskal's Algorithm

Algorithm 9.7. MST-Kruskal (G, w)

1. A <-- ϕ
2. for each vertex $v \in V(G)$
3. do MAKE-SET (V)
4. sort the edges of E into non-decreasing order by weight w.
5. For each edge $(u, v) \in E$, taken in non-decreasing order by weight.
6. do if FIND-SET $(u) \neq$ FIND-SET (v)
7. then A <-- $A \cup \{(u, v)\}$
8. UNION (u, v)
9. Return A.

Kruksal's algorithm, initially initialize the set A to the empty set and create $|V|$ trees or sets, each containing one vertex. The edges in E are sorted into non-decreasing order by weight and may be stored in a queue in line 4.

The for loop of line 5-8, checks for an edge from the queue *i.e.* (u, v), whether the endpoints u and v belongs to the same set, if they are, then the edge (u, v) must be discarded, otherwise the edge must be added to set A and the verteces of the two sets are merged together in line 8.

9.5.1.3. Analysis of Kruskal's Algorithm

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structures initially the set A takes $O(1)$ time; and the time to sort the edges in line 4 is $O(E \log E)$. In loop 5-8 every edge' end points are checked using $O(E)$ find-set and UNION operations on disjoint sets.

In loop 2-3, there are $|V|$ make-set operations.

$$\text{Hence the total time} = T_{\text{step }1} + T_{\text{step }2\text{-}3} + T_{\text{step }4} + T_{\text{step }5\text{-}8}$$
$$= O(1) + O(v) + O(E \log E) + O(E)$$

As G is assumed to be connected, hence we have

$$|E| \geq |V| - 1$$

therefore $\quad O(V) + O(E) = O(E)$

Hence the total time $\cong O(E) + O(E \log E) \cong O(E \log E)$

If $|E| < |V|^2$, then $\log E = O(\log V)$

then the total time $\cong O(E \log V)$.

9.5.2. PRIM Algorithm

The PRIM algorithm is used to the find solution for the minimum spanning tree problem. The steps of this algorithm are presented as follows :

Step 1 : Represent the distance network in matrix form (if there is no arc between node i and node j, then set the distance between them to infinity/a very high value. Also, set the diagonal entries of the distance matrix to infinity/a very high value.)

Step 2 : Let Q = [Null set} which is the set of selected row numbers of the matrix.

Step 3 : Select row 1 and include it in Q. Then, delete column 1 of the matrix.

Step 4 : Find the minimum of the unselected values among the rows in Q and select it by marking a square around it (in case of tie, break, it randomly).

Step 5 : Identify the corresponding column number (K) and then include it in Q.

Step 6 : Delete column K of the matrix.

Step 7 : Check whether all the column are deleted.

If yes, go to step 8, otherwise go to step 4.

Step 8 : Show the arcs in the spanning tree corresponding to the cells of the matrix marked with squares by thick lines.

Step 9 : Find the sum of all the distance values marked with squares. This is the minimized total length of the arcs to connect all the nodes of the network as per the minimum spanning tree concept.

Step 10 : Stop.

Example 9.9. Consider the distance network shown in Fig. 9.36, which is reproduced below in Fig. 9.38. Find the minimum spanning tree of the this network using the PRIM algorithm.

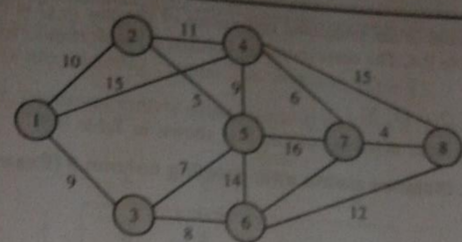


Fig. 9.38. Distance network (Example 9.9)

Solution.

Step 1 : The matrix form of the distance network is shown in Table 9.3.

Table 9.3. Distance matrix (Example 9.9)

	Node j								
		1	2	3	4	5	6	7	8
Node i	1	–	10	9	15	∞	∞	∞	∞
	2	10	–	∞	11	5	∞	∞	∞
	3	9	∞	–	∞	7	8	∞	∞
	4	15	11	∞	–	9	∞	6	15
	5	∞	5	7	9	–	14	16	∞
	6	∞	∞	8	∞	14	–	10	12
	7	∞	∞	∞	6	16	10	–	4
	8	∞	∞	∞	15	∞	12	4	–

Step 2 : Q = {Null set}.

Step 3 : Q = {1}. The matrix after deleting column 1 is shown in Table 9.4. In the following tables, each of the rows in Q is indicated by *.

Table 9.4. Distance matrix after deleting column 1 (Example 9.9)

		Node j						
		2	3	4	5	6	7	8
*	1	10	$\boxed{9}$	15	∞	∞	∞	∞
	2	–	∞	11	5	∞	∞	∞
	3	∞	–	∞	7	8	∞	∞
	4	11	∞	–	9	∞	6	15
Node i	5	5	7	9	–	14	16	∞
	6	∞	8	∞	14	–	10	12
	7	∞	∞	6	16	10	–	4
	8	∞	∞	15	∞	12	4	–

Step 4 : The minimum of the undeleted cell values of the row in Q is 9. So, it is marked with a square as shown in Table 9.4. The corresponding column (k) with respect to the minimum distance is 3.

Step 5 : $K = 3$ and $Q = \{1, 3\}$.

Step 6 : The matrix after deleting column 3 is shown in Table 9.5.

Table 9.5. Distance matrix after deleting column 3 (Example 9.9)

		Node j → 2	4	5	6	7	8
*	1	10	15	∞	∞	∞	∞
	2	–	11	5	∞	∞	∞
*	3	∞	∞	$\boxed{7}$	8	∞	∞
	4	11	–	9	∞	6	15
Node i	5	5	9	–	14	16	∞
	6	∞	∞	14	–	10	12
	7	∞	6	16	10	–	4
	8	∞	15	∞	12	4	–

Step 7 : All columns are not deleted. So, go to step 4.

Step 4 : The minimum of the undeleted cell values of the rows in Q is 7, it is marked with a square as shown in Table 9.7.

Step 5 : $K = 5$ and $Q = \{1, 3, 5\}$

Step 6 : The matrix after deleting column 5 is shown in Table 9.6.

Table 9.6. Distance matrix after deleting column 5 (Example 9.9)

		Node j → 2	4	6	7	8
*	1	10	15	∞	∞	∞
	2	–	11	∞	∞	∞
*	3	∞	∞	8	∞	∞
	4	11	–	∞	6	15
Node i *	5	$\boxed{5}$	9	14	16	∞
	6	∞	∞	–	10	12
	7	∞	6	10	–	4
	8	∞	15	12	4	–

Step 7 : All columns are not deleted. So, go to step 4.

Step 4 : The minimum of the undeleted cell values of the rows in Q is 5. So, it is marked with a square as shown in Table 9.6.

Step 5: $K = 2$ and $Q = \{1, 3, 5, 2\}$

Step 6 : The matrix after deleting column 2 is shown in Table 9.7.

Table 9.7. Distance matrix after deleting column 2 (Example 9.9)

			Node j			
			4	6	7	8
	*	1	15	∞	∞	∞
	*	2	11	∞	∞	∞
		3	∞	$\boxed{8}$	∞	∞
		4	–	∞	6	15
Node i	*	5	9	14	16	∞
		6	∞	–	10	12
		7	6	10	–	4
		8	15	12	4	–

Step 7 : All columns are not deleted. So, go to step 4.

Step 4: The minimum of the undeleted cell values of the rows in Q is 8. So, it is marked with a square as shown in Table 9.7.

Step 5 : $K = 6$ and $Q = \{1, 3, 5, 2, 6\}$.

Step 6 : The matrix after deleting column 6 is shown in Table 9.8.

Table 9.8. Distance matrix after deleting column 6 (Example 9.9)

			Node j		
			4	7	8
	*	1	15	∞	∞
	*	2	11	∞	∞
		3	∞	∞	∞
		4	–	6	15
Node i	*	5	$\boxed{9}$	16	∞
	*	6	∞	10	12
		7	6	–	4
		8	15	4	–

Step 7 : All columns are not deleted. So, go to step 4.

Step 4 : The minimum of the undeleted cell values of the rows in Q is 9. So, it is marked with square as shown in Table 9.8.

Step 5 : $K = 4$ and $Q = \{1, 3, 5, 2, 6, 4\}$.

Step 6 : The matrix after deleting column 4 is shown in Table 9.9.

Table 9.9. Distance matrix after deleting column 4 (Example 9.9)

		Node j	
		7	8
*	1	∞	∞
*	2	∞	∞
*	3	∞	∞
*	4	$\boxed{6}$	15
Node i *	5	16	∞
*	6	10	12
	7	–	4
	8	4	–

Step 7 : All column are not dleted. So, go to step 4.

Step 4 : The minimum of the undeleted cell values of the rows in Q is 6. So, it is marked with a square as shown in Table 9.9.

Step 5 : $K = 7$ and $Q = \{1, 3, 5, 2, 6, 4, 7\}$

Step 6 : The matrix deleting column 7 is shown in Table 9.10.

Table 9.10. Distance matrix after deleting column 7 (Example 9.9)

		Node j
		8
*	1	∞
*	2	∞
*	3	∞
*	4	15
Node i *	5	∞
*	6	12
*	7	$\boxed{4}$
	8	–

Step 7 : All columns are not deleted. So, go to step 4.

Step 4 : The minimum of the undeleted cell values of the rows in Q is 4. So, it is marked with a square as shown in Table 9.10.

Step 5 : $K = 8$ and $Q = \{1, 3, 5, 2, 6, 4, 7, 8\}$

Step 6 : After deleting column 8, the entire matrix is deleted.

Step 7 : All the coulmn are deleted. So, go to step 8.

Step 8 : The minimum spanning tree is shown in Figure 9.39 and the corresponding minimum total length of the arcs is 48.

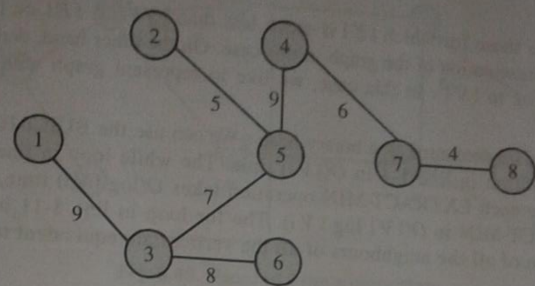


Fig. 9.39. Minimum spanning tree (Example 9.9).

9.5.2.1. Prim's Algorithm

Algorithm 9.8. MST-PRIM (*G, w, r*)

1. For each $u \in v(G)$
2. do key $[u] \leftarrow \infty$
3. $\pi[u] \leftarrow$ Nil
4. Key $[r] \leftarrow 0$
5. Set $q \leftarrow v(G)$
6. While $(q \neq \phi)$
7. do $u \leftarrow$ EXTRACT-MIN (q)
8. for each $v \in$ Adj$[u]$
9. do if $v \in q$ and $w(u, v) <$ key $[v]$
10. then $\pi[v] \leftarrow u$
11. key $[v] \leftarrow w(u, v)$

Prim's algorithm, initially in line 1-5. Set the key or distance of each vertex to ∞ (except the root r, whose key is set to 0), set the parent π of each node to Nil and initialize the minimum priority queue 'q' to contain all the vertices.

Loop 6-11 repeatly extract a vertex from the priority queue, which is having the maximum priority and compare the existing distance of each vertex v, with the new distance found out using the following :

(*a*) Suppose there is an edge (u, v) then existing distance is represented by key $[v]$ while the new distance can be determined using the vertex u *i.e.* key $[u] + w(u, v)$ and whichever is less or minimum assign that value to key $[v]$.

9.5.2.2. Analysis of Prim's Algorithm

The performance of Prim's algorithm depends on how we choose to implement the priority queue q.

Definitions

Sparse graphs are those for which $|E|$ is much less than $|V|^2$ *i.e.* $|E| << |V|^2$, we preferred the adjacency list representation of the graph in this case. On the other hand, dense graph are those for which $|E|$ is close to $|V|^2$. In this case, we like to represent graph with adjacency matrix representation.

(*a*) When a 'q' is implemented as a binary heap : We can use the BUILT-HEAP procedure to perform the initialization in line 1-4 in $O(|V|)$ time. The while loop in line 6-11 is executed $|V|$ times, and since each EXTRACT-MIN operation takes $O(\log |V|)$ time, the total time for all calls to EXTRACT-MIN is $O(|V| \log |V|)$. The for loop in line 8-11 is executed $O(|E|)$ times. Since the sum of all the neighbours of all the vertices are equivalent to number of edges in the graph *i.e.*

$$\sum_{v \in V(G)} Adj(v) = O(E)$$

The assignment in line 11 involves an implicit. DECREASE-KEY operation, which takes $O(\log |V|)$ time.

Thus the total time for prim's algorithm using binary heap

$$= O(|V| \log |V|) + O(|E| \log |V|)$$

$$= O(|V| \log |V| + |E| \log |V|)$$

(*b*) When 'q' is implemented using Fibonacci heap : In this case, the performance of the DECREASE-KEY operation implicitly called in line 11 will be improved. Instead of taking $O(\log |V|)$ time with a Fibonacci heap of $|V|$ elements, DECREASE-KEY takes $O(1)$ time, therefore the total time $= O(|V| \log |V| + |E|)$.

(*c*) When graph $G = (V, E)$ is sparse : $|E| = \Theta(|V|)$ from above description, it can be seen that both versions of prim's algorithm will have the running time $O(|V| \log |V|)$. Therefore, the Fibonacci heap implementation will not make prim's asymptotically faster for sparse graph.

(*d*) When graph $G = (V, E)$ is dense : From above description. We can see that prim's algorithm with binary heap will take $O(|V|^2 \log |V|)$ time whereas with fibonacci heap, will take $O(|V|^2)$ time. Therefore, fibonacci heap implementation doesn't make prim's algorithm asymptotically faster for dense graph.

Example 9.10. Prove the correctness of PRIM's Algorithm ?

Solution. Prim's algorithm always yields a MST. We can prove it by induction. Let T_0 be a part of any MST which consists of a single vertex. Assume that T_{i-1} is a part of MST. We need to prove that T_i, generated by T_{i-1} by Prim's algorithm is a part of a MST. Assume that no MST of the graph can contain T_i.

Let $e_i = (u, v)$ be a minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i. By out assumption, if we add e_i to T, a cycle must be formed. In addition to e_i, the cycle must contain another edge (v', u'). If we delete the edge e_k (v', u'), then we obtain another spanning tree. Now we have $w_{ei} \leq w_{ek}$. So, weight of new spanning tree is less

than or equal to T. New spanning tree is a minimum spanning tree which contradicts the assumption that no minimum spanning tree contains T_i.

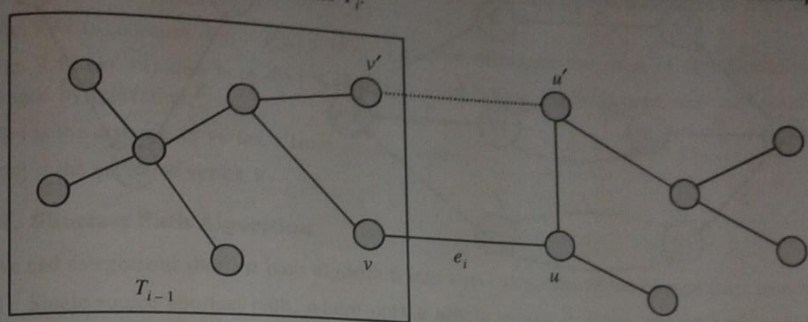


Fig. 9.40. Proof of Prim's algorithm.

9.6. SHORTEST PATH

Suppose G be a weighted directed graph where a minimum labeled $w(u, v)$ associated with each edge (u, v) in E, called weight of edge (u, v). These weights represent the cost to traverse the edge. A path from vertex u to vertex v is a sequence of one or more edges.

$$< (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n) > \text{ in } E(G)$$

where $\quad u = v_1$ and $v = v_n$

The cost (or length or weight) of the path P is the sum of the weights of edges in the sequence.

The shortest-path weight from a vertex $u \in V$ to a vertex $v \in V$ in the weighted graph is the minimum cost of all paths from u to v. If there exists no such path from vectex u to vertex v then the weight of the shortest path is ∞.

9.6.1. Variant of Shortest Path Problems

(1) **Single-destination shortest paths problem :** Find a shortest path to a given destinatio vertex t from each vertex v. By reversing the direction of each edge in the graph, we ca reduce this problem to a single-source problem.

(2) **Single-pair shortest path problem :** Find a shortest path from u to v for given vertices and v. If we solve the single source problem with source vertex u, we solve this proble also. Moreover, no algorithm for this problem are known that run asymptotically fast than the best single-source algorithm in worst case.

(3) **All-pairs shortest path problem :** Find a shortest path from u to v for every pair vertices u and v. Although this problem can be solved by running a single source algorit once for each vertex, it can usually be solved faster.

9.6.2. Negative Weight Edges

The negative weight cycle is a cycle whose total is negative. No path from starting vertex S a vertex on the cycle can be a shortest path. Since a path can run around the cycle many times get any negative cost desired, in other words, a negative cycle invalidates the notion of dista based on edge weights.

Relaxation of an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex is shown within the vertex.

Fig. 9.42 (*a*),because $d[v] > d[u] + w(u, v)$ prior to relaxation the value of $d[v]$ decreases.

Fig. 9.42 (*b*), because here $d[v] \leq d[u] + w[u, v]$ before the relaxation step, and so $d[v]$ is unchanged by relaxation.

$d[v]$ is the distance of vertex v from source vertex.

$\pi[v]$ is the parent of vertex V.

9.6.4. Shortest Path Algorithm

We had categorized shortest path algorithm into two categories depending on their sources *i.e.*

(1) Single source shortest path, where only a single source is given and we have to find the shortest path of each vertex V from the source S. There are two basic algorithms to find single source shortest paths are :

(*a*) Bellman-Ford algorithm.

(*b*) Dijkstra's algorithm

	Complexity	Description
Bellman Ford	$O(VE)$	Supports negative weight edges as well as negative weight cycles in the graph.
Dijkstra	$O(E \log V)$	Supports neither negative edges nor negative weight cycles.

(2) All pair shortest path, where we have to find a shortest path from u to v for every pair of vertices u and v, using on algorithm known as Floyd Warshall algorithm.

	Complexity	Description
Floyd Warshall	$O(\lvert V \rvert^3)$	Supports only negative weight edges but not negative weighted cycles in graph.

9.7. THE BELLMAN FORD ALGORITHM

Given a weighted, directed graph $G = (V, E)$ with source S and weight function $w : E \rightarrow R$, the Bellman Ford algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the source. If there is such a cycle with negative total, the algorithm indicates that no solution exists.

The algorithm uses relaxation progressively decreasing an estimate $d[v]$ on the weight of a shortest path from source S to each vertex $v \in V$ until it achieves the actual shortest path weight $\delta(S, V)$.

To find the matrix for reflexive transitive closure, just replace infinity by zero, non-zero by one and all zero by one in matrix $D^{(n)}$.

EXERCISES

1. What is graphs. Explain how the graph are represented ?
2. Difference between BFS and DFS ?
3. Discuss the application of DFS.
4. What is parenthesis structure.
5. What is white-path theorem.
6. Difference between Prim's and Kruskal algorithms for finding maximum spanning tree ?
7. Why Dijkstra algorithm does not support negative weighted edges in a graph.
8. What is shortest path ? Explain all its techniques with their complexities.

QUESTIONS WITH ANSWERS

Q. 1. Find the shortest path between all pairs of nodes in the following graph.

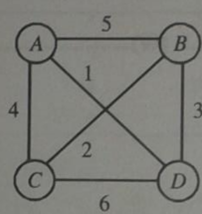
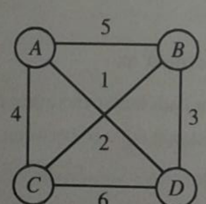


Fig. 9.49

Ans. $A^k(i, j) = \min\limits_{k=1,2,3,4} \left\{A^{k-1}(i,k) + A^{k-1}(k,j), C(i,j)\right\}$ where $n = 4$, no. of nodes



$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 4 & 1 \\ 2 & 5 & 0 & 2 & 3 \\ 3 & 4 & 2 & 0 & 6 \\ 4 & 1 & 3 & 6 & 0 \end{array}$$

Fig. 9.50

for $k = 1$

1 1 $A^1(1, 1) = \min [A^0(1, 1) + A^0(1, 1), C(1, 1)] = \min [0 + 0, 0] = 0$

1 2 $A^1(1, 2) = \min [A^0(1, 1) + A^0(1, 2), C(1, 2)] = \min [0 + 5, 5] = 5$

1 3 $A^1(1, 3) = \min [A^0(1, 1) + A^0(1, 3), C(1, 3)] = \min [0 + 4, 4] = 4$

1 4 $A^1(1, 4) = \min [A^0(1, 1) + A^0(1, 4), C(1, 4)] = \min [0 + 1, 1] = 1$

Similarly, for other values need to compute likewise, for those values that are changed have been done as

3 4 $A^1(3, 4) = \min [A^0(3, 1) + A^0(1, 4), C(3, 4)] = \min [4 + 1, 6] = 5$

4 3 $A^1(4, 3) = \min [A^0(4, 1) + A^0(1, 3), C(4, 3)] = \min [1 + 4, 6] = 5$

$$A^1 \begin{array}{|cccc|} \hline 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \\ \hline \end{array}$$

Thus, A^1 matrix is obtained will be the input for A^2.

In this iteration values of matrix doesn't change but needs to compute for next iteration *i.e.*, k = 2.

$$A^2 \begin{array}{|cccc|} \hline 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \\ \hline \end{array}$$

In this iteration values of matrix doesn't change but needs to compute for next iteration *i.e.*, k = 3

$$A^3 \begin{array}{|cccc|} \hline 0 & 5 & 4 & 1 \\ 5 & 0 & 2 & 3 \\ 4 & 2 & 0 & 5 \\ 1 & 3 & 5 & 0 \\ \hline \end{array}$$

Finally, A^3 is the resultant values for all pair shortest path.

Q. 2. What is a spanning tree ? Show that a simple graph is connected if it has a spanning tree.

Ans. A spanning tree of a connected graph is a spanning subgraph that is a tree. A spanning tree is not unique the graph is a tree. Spanning trees have applications to design of communication networks.

A spanning tree of an undirected graph of n nodes is a set of $n - 1$ edges that connects all nodes. Consider the following connected, undirected graph and notice the number of edges. Also, each node is reachable from every other node. A spanning tree can be obtained by using either DFS or BFS.

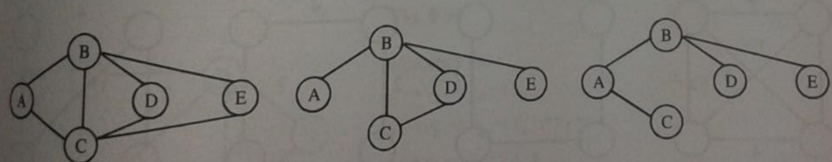


Fig. 9.51. An undirected graph with its DFS and BFS spanning trees.

For example : A is connected to B and E through path A → B → E, and C and D by path A → B → C → D.

Q. 3. What is a minimum spanning tree ? What are its applications ?

Ans. A connected graph is a tree if and only if it has n vertices and $n - 1$ edges. Assume there is an undirected, connected graph G. A spanning tree is a subgraph of G, is a tree, and contains all the vertices of G. A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weight of its edges) is at a minimum. Following is an undirected, weight graph with two MSTs, each with a cost of 6 :

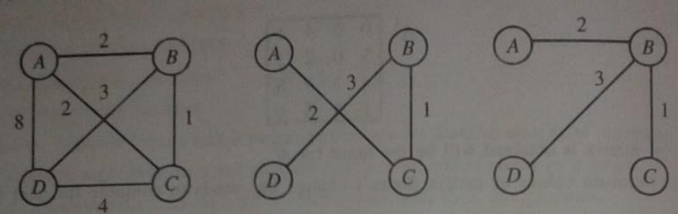


Fig. 9.52

Minimum spanning tree have many applications. Some are :

(*i*) Building cable networks that join n locations with minimum cost.

(*ii*) Building road network that joins n cities with minimum cost.

(*iii*) Optimizing computer networks.

(*iv*) Obtaining an indpendent set of circuit equations for an electrical network.

(*v*) In pattern recognition, MST can be used to find noisy pixels.

Q. 4. Prove that the edge with the smallest weight will be part of every minimum spanning tree.

Ans. Edge with the smallest weight will be part of every minimum spanning tree.

Let u-v be the smallest weight edge. Pick any minimum spanning tree T. If the edge is in T, we are done. So assume that the edge is not in T. But then when we add u-v to T, we get a cycle. Pick any edge on the cycle other then u-v and remove the edge. The weight of the resulting subgraph is less than or equal to the weight of T. Moreover, the graph is a tree since it is connected and has $n - 1$ edges (n : number of vertices in the graph). Kruskal's algorithm builds MST. The first edge that we add to T will be the smallest weight edge in the graph. Hence, the smallest weight will be part of every MST.

It is observed from the following figure that the minimum weight edge (weight = 2) of the graph is included in three minimum spanning trees.

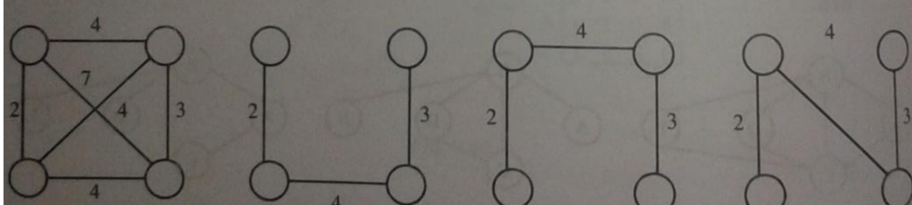


Fig. 9.53

Q. 5. Write Prim's algorithm under the assumption that the graph are represented by adjacency lists. Analyze the computing time and space requirements of this new version of Prim's algorithm using adjacency lists.

Ans. Let $G = (V, E)$ which is represented by an adjacency list Adj.

d is an array of size |V|. Each $d[i]$ contains the shortest distance for vertex i.

Q is a priority queue of UNKNOWN vertices.

p is an array of size |V|. Each $p[i]$ contains the parent of vertex i.

s is the source vertex.

```
PRIM(G, s)
1. Initialize d[s] with 0, p[s] with 0, and all other d[i] (i ! = s) with a positive infinitty and p[i]
   (i ! = s) with 0.
2. Q ← V                          // initialize Q with all vertices as UNKNOWN.
3. While Q not empty do
4.     u ← DeleteMin (Q)          // Q is modified
5.     Mark u as KNOWN            // Dequeing u is the same as marking it as KNOWN
6.     For each vertex v in Adj[u] do
7.         If v is UNKNOWN and d[v] > weight (u, v), then
8.             d[v] = weight (u, v)   // update with smaller weight
9.             p[v] = u               // update v's parent as v
10.        Endif
11.    Endfor
12. Endwhile.
```

If a graph is represented by adjacency linked lists and priority queue is implemented as a min-heap, running time of the algorithm will be in $O(|E| \log |V|)$. Example 4.21 illustrates this algorithm.

Q. 6. Obtain the minimum spanning tree of the following graph, using Prim's algorithm under the assumption that the graphs are represented by adjacency lists.

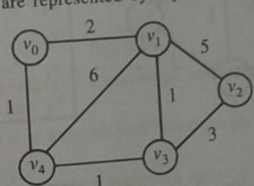
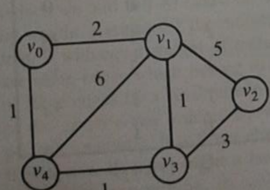


Fig. 9.54

Ans.



Adjacency list :

$v_0 \to v_1 \to v_4 \to x$

$v_1 \to v_0 \to v_2 \to v_3 \to v_4 \to x$

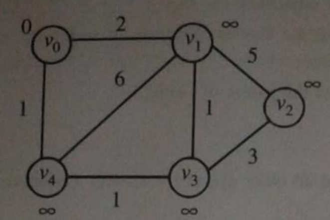
$v_2 \to v_1 \to v_3 \to x$

$v_3 \to v_1 \to v_2 \to v_4 \to x$

$v_4 \to v_0 \to v_1 \to v_3 \to x$

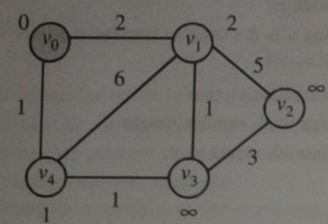
x : null

v_0 is the source vertex, and $d[i]$ for each vertex i is indicated at each node. The stages of Prim's algorithm are shown below :



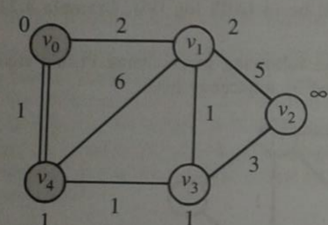
v	known	$d[v]$	$p[v]$
v_0	F	0	0
v_1	F	∞	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0

$Q = \{ v_0, v_1, v_2, v_3, v_4\}$



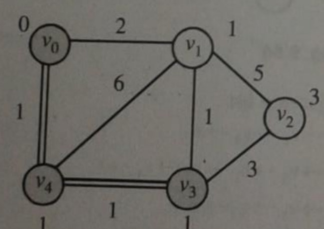
v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	F	2	v_0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	1	v_0

$Q = \{v_4, v_1, v_2, v_3\}$



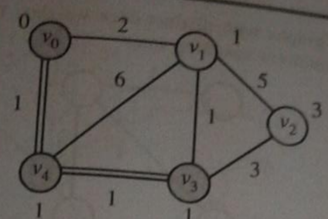
v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	F	2	v_3
v_2	F	∞	0
v_3	F	1	v_4
v_4	T	1	v_0

$Q = \{v_3, v_1, v_2\}$



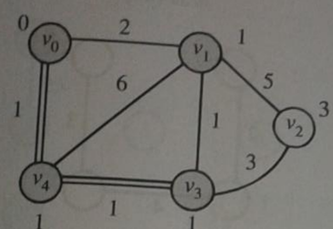
v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	F	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{v_1, v_2\}$



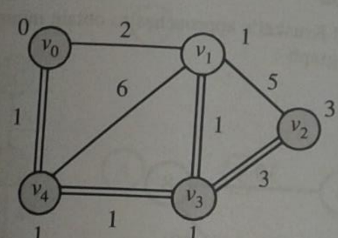
v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	F	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{v_1, v_2\}$



v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	T	1	v_3
v_2	F	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{v_2\}$



v	known	$d[v]$	$p[v]$
v_0	T	0	0
v_1	T	1	v_3
v_2	T	3	v_3
v_3	T	1	v_4
v_4	T	1	v_0

$Q = \{\ \}$

Fig. 9.55.

Q. 7. Prove that if the weights on edges of a connected undirected graph are distinct, then there is a unique minimum spanning tree.

Ans. If the weights on edges of a connected undirected graph are distinct, then there is a unique minimum spanning tree.

If each edge has a distinct weight then there will only be one, unique minimum spanning tree. This can be proved by induction or contradiction. Say we have an algorithm that finds an MST (which we will call A) based on the structure of the graph and the order of the edges when ordered by weight. Assume for the moment that this MST is not unique and that there is another spanning tree, B with equal weight. If there are n vertices in the graph, then each tree has $n - 1$ edges. There are some edges which belong to B but not to A. What happens if we decrease the weight of one of these edges by a small amount e so that we do not change the overall ordering of all the edges of the graph when ordered by weight ? (This is possible because all weights are separated by positive amounts). It will not change the result of our algorithm, which still gives tree A. But tree B will now have a weight e less than what it has before, which means that A is not minimal, contrary to assumption. Because of this contradiction, we conclude that the assumption that there can be a second MST weas false.

Consider two different undirected connected graphs with distinct edge weights. The following figure shows two graphs with their unique minimum spanning trees.

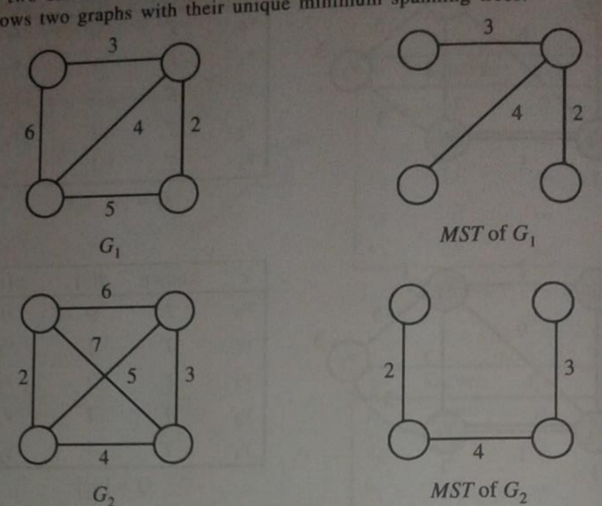


Fig. 9.56

Q. 8. What is the basic difference between Prim's and Kruskal's approaches to obtain minimum spanning trees ? Illustrate the results for the following graph :

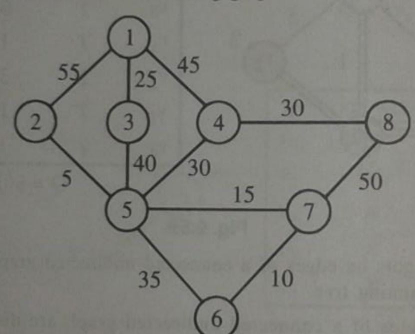


Fig. 9.57

Ans.

Prim's Algorithm	Kruskal's Algorithm
1. Build MST one vertex at a time.	1. Construct MST one edge at a time.
2. Start with one vertex tree T.	2. Sor the edges by weight.
3. Add the edge of minimum weight among those with one vertex in T and the other not in T.	3. Build a spanning forest (that eventually becomes a tree) by adding the edge of minimum weight which when added to those already chosen does not form a cycle.

The following figure shows the stages of Prim's MST algorithm (starting vertex is 1). Total minimum cost is 155.

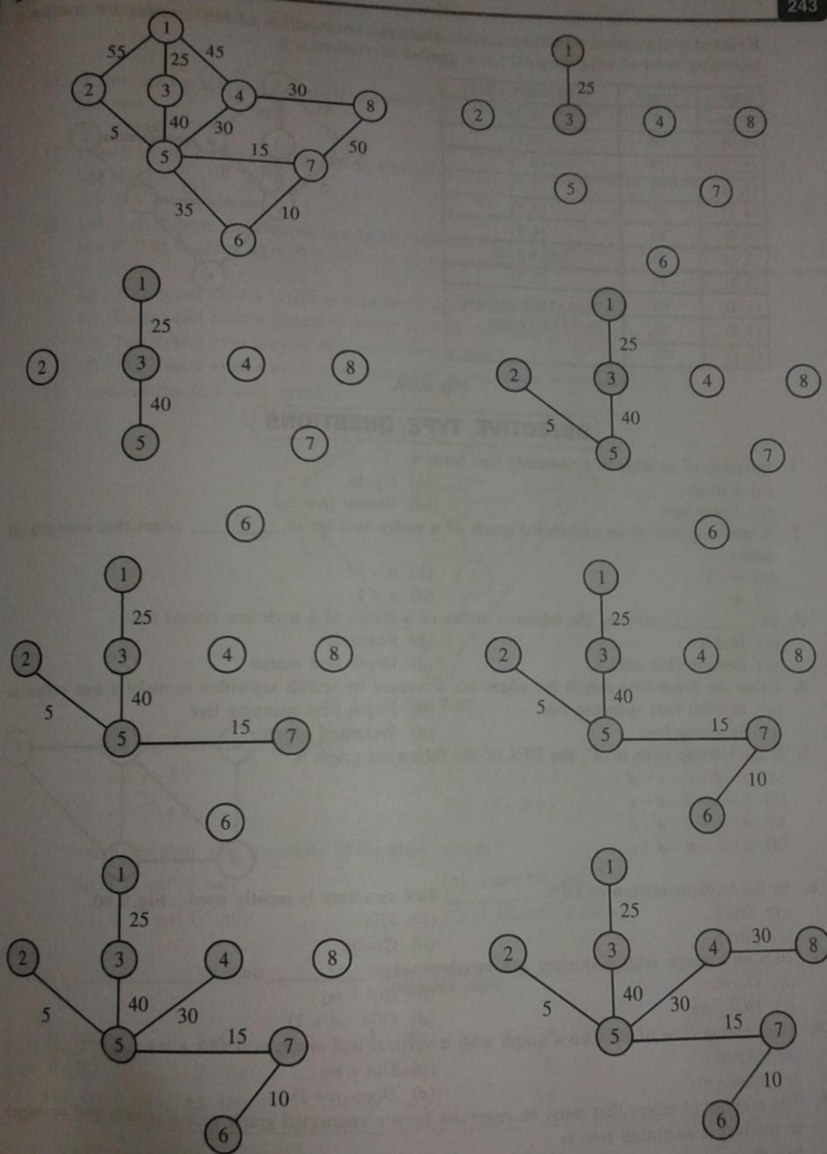


Fig. 9.58.

Kruskal's Algorithm : Following table gives the construction of MST (edges are aranged in ascending order of edge weights) : n = number of vertices = 8.

Edge	weight	Edge added in MST
(2, 5)	5	(2, 5)
(6, 7)	10	(6, 7)
(5, 7)	15	(5, 7)
(1, 3)	25	(1, 3)
(4, 5)	30	(4, 5)
(4, 8)	30	(4, 8)
(5, 6)	35	Forms cycle
(3, 5)	40	(3, 5)
(1, 4)	45	Stop : MST contains (n – 1) = 7 edges.
(7, 8)	50	
(1, 2)	55	

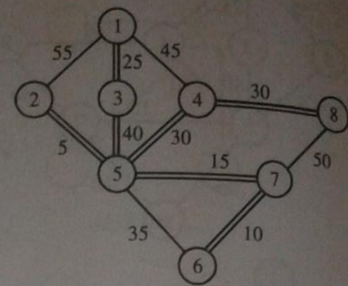


Fig. 9.59.

OBJECTIVE TYPE QUESTIONS

1. Insertion of an edge to a spanning tree form a
   (a) Circuit (b) Cycle
   (c) Open path (d) Binary tree
2. A spanning tree of an undirected graph of n nodes is a set of ________ edges that connects all nodes.
   (a) $n - 1$ (b) $n - 2$
   (c) n (d) $n + 1$
3. In ________ traversal the adjacent nodes of a nodes of a node are visited first.
   (a) Inorder (b) Postorder
   (c) Breadth first search (d) Depth first search
4. Using the depth first search the edges are traversed by search algorithm to make a tree called as
   (a) Breadth first spanning tree (b) Depth first spanning tree
   (c) Spanning tree (d) Weighted graph
5. If the starting node is 'a', the DFS of the following graph is
   (a) $a - b - c - e - d$
   (b) $b - e - d - a - c$
   (c) $b - e - a - d - c$
   (d) $a - c - b - d - e$

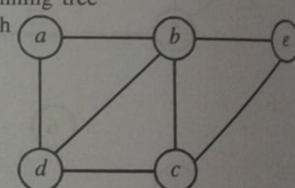


Fig. 9.60.

6. In the implementation of DFS ________ data structure is mostly used.
   (a) Stack (b) File
   (c) Array (d) Queue
7. DFS on a graph with n vertices and m edges takes ________ time.
   (a) $O(mn)$ (b) $O(n + m)$
   (c) $O(m - n)$ (d) $O(m - n + 1)$
8. The running time of BFS on a graph with n vertices and m edges is $O(n + m)$.
   (a) $O(mn)$ (b) $O(n + m)$
   (c) $O(m - n)$ (d) $O(m - n + 1)$
9. The number of edges that must be removed from a connected graph with n vertices and m edges to produce a spanning tree is
   (a) m (b) $n - m$
   (c) $m - n + 1$ (d) $m - n$

10. The minimum number of edges in a connected cyclic graph on n vertices is
    - (*a*) $n - 1$
    - (*b*) n
    - (*c*) $n + 1$
    - (*d*) None of these
11. A directexd graph has a cycle if and only if its depth-first search reveals a _____ edge
    - (*a*) tree
    - (*b*) forward
    - (*c*) cross
    - (*d*) back
12. Which of the following is useful in traversing a graph by breadth-first search ?
    - (*a*) stack
    - (*b*) set
    - (*c*) list
    - (*d*) queue
13. Let T be a depth first search tree in an undirected graph G. Vertices u and v are leaves of this tree T. The degrees of both u and v in G are at least 2. Which one of the following statements is true ?
    - (*a*) There must exist a vertex w adjacent to both u and v in G.
    - (*b*) There must exist a vertext w whose removal disconnects u and v in G.
    - (*c*) There must exist a cycle in G containing u and v.
    - (*d*) There must exist a cycle in G containing u and all its neighbours in G.
14. Consider the following graph :

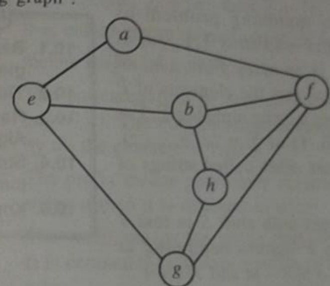


Fig. 9.61.

Among the following sequences

I. $a\ b\ e\ g\ h\ f$ II. $a\ b\ f\ e\ h\ g$

III. $a\ b\ f\ h\ g\ e$ IV. $a\ f\ g\ h\ b\ e$

which are depth first traversals of the above graph ?

- (*a*) I, II and IV only
- (*b*) I and IV only
- (*c*) II, III and IV only
- (*d*) I, III and IV only

ANSWERS

1. (*a*)	2. (*b*)	3. (*d*)	4. (*d*)	5. (*b*)
6. (*a*)	7. (*b*)	8. (*d*)	9. (*d*)	10. (*b*)
11. (*a*)	12. (*b*)	13. (*d*)	14. (*d*)	

□□□